Pro Git

Scott Chacon*

2014-11-28

*지금 보시는 문서는 책 「Pro Git」의 PDF 파일입니다. 본 문서는 크리에이티브 커먼즈 저작자표시-비영리-동일조건변경허락 3.0 (Creative Commons Attribution-Non Commercial-Share Alike 3.0) 조건에 따라 이용할 수 있습니다. 이 책을 통해 Git을 즐겁게 배울 수 있기를 바랍니다. 출판된 책을 Amazon 웹사이트http://tinyurl.com/amazonprogit에서 구입하시면 원저자인 Scott Chacon과 Apress 출판사에게 큰 도움이 됩니다.
도서출판 인사이트를 통해서 한국어판이 2013년 4월 19일에 출판되었습니다. 한국어 번역에 대한 자세한 정보는 https://github.com/progit/progit/tree/master/ko에서 확인해 주세요.

목차

1장

시작하기

이 장에서는 Git을 처음 접하는 사람에게 필요한 내용을 다룬다. 버전 관리 도구에 대한 약간의 배경지식, Git의 특징, Git을 설치하는 법 그리고 Git을 시작하기에 앞서 필요한 설정을 하는 방법을 설명한다. 이 장을 다 읽고 나면 Git의 탄생 배경과 Git이 사용되는 이유를 이해하고, Git을 시작하기 위한 준비가 되어있을 것이다.

1.1 버전 관리란?

버전 관리란 무엇이며, 왜 이것을 알아야 할까? 버전 관리 시스템은 파일의 변화를 시간에 따라 기록하여 과거 특정 시점의 버전을 다시 불러올 수 있는 시스템이다. 이 책에는 소프트웨어의 소스 코드를 버전 관리하는 예만 나오지만 실제로는 모든 컴퓨터 파일이 버전 관리의 대상이 될 수 있다.

이미지나 레이아웃을 수정할 때마다 각각의 형태를 모두 보존하고 싶은 그래픽 디자이너나 웹 디자이너라면 버전 관리 시스템(Version Control System; VCS)을 사용하는 것이 현명할 수 있다. VCS를 사용하면 개별 파일 혹은 프로젝트 전체를 이전 상태로 되돌리거나 시간에 따른 변경 사항을 검토할 수 있으며, 문제가 되는 부분을 누가 마지막으로 수정했는지, 누가 언제 이슈를 만들어냈는지 등을 알 수 있다. 또한 파일을 잃어버리거나 무언가 잘못되어도 대개 쉽게 복구할 수 있다. 그리고 이 모든 장점을 누리는 데는 큰 노력이 들지 않는다.

1.1.1 로컬 버전 관리 시스템

대부분의 사람들이 버전 관리를 위해 쓰는 방법은 파일을 다른 디렉토리에 복사하는 것이다(똑똑한 사람이라면 디렉토리 이름에 시간을 넣을 것이다). 이 방법은 간단하고 자주 사용되는 방법이지만 실수가 발생하기 쉽다. 어느 디렉토리에서 작업하고 있었는지 잊어버리고 엉뚱한 파일을 덮어쓰거나 의도하지 않았던 위치로 복사할 수도 있다.

이 문제를 해결하기 위해 오래전에 프로그래머들은 간단한 데이터베이스에 파일의 변경 사항을 기록하는 로컬 버전 관리 시스템을 만들었다(그림 1-1 참조).

유명했던 VCS 도구들 중 현재에도 널리 쓰이는 것으로 RCS라 불리는 시스템이 있다. 그 예로 Mac OS X 운영체제에서는 개발 도구를 설치하면 RCS가 딸려온다. RCS의 기본적인 동작 방식은 각 리비전들 간의 패치 세트(patch set)라고 하는 데이터의 차이점들을 특별한 형식의 파일에 저장, 특정 시점의 파일 내용을 보고 싶을 때 해당 시점까지의 패치들을 모두 더하여 파일을 만들어내는 것이다.

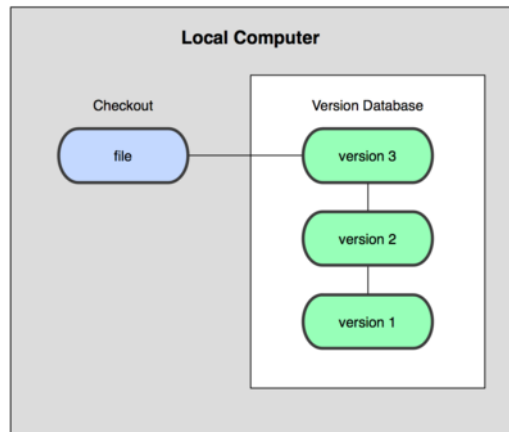



그림 1.1: 로컬 버전 관리 다이어그램

1.1.2 중앙집중식 버전 관리 시스템

또 다른 문제는 시스템 외부에 있는 개발자들과 함께 작업하는 것이다. 중앙집중식 버전 관리 시스템(Centralized Version Control System; CVCS)은 이 문제를 해결하기 위해 개발됐다. CVS, Subversion, Perforce와 같은 시스템들이 여기에 속한다. CVCS에서는 버전 관리되는 모든 파일을 저장하는 하나의 서버와, 이 중앙 서버에서 파일들을 가져오는(checkout) 다수의 클라이언트가 존재한다. 오랫동안 사용된 이 방식은 지금까지도 버전 관리의 대표적인 방식이다(그림 1-2 참조).

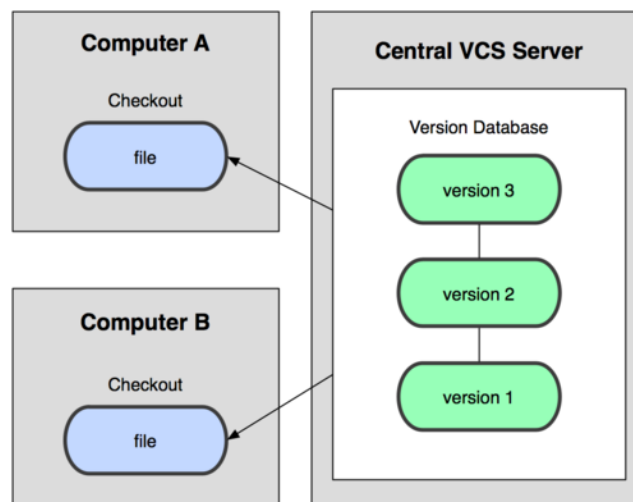



그림 1.2: 중앙집중식 버전 관리 다이어그램

CVCS는 로컬 VCS에 비해 장점이 많다. 누구나 다른 사람들이 무엇을 하고 있는지 알 수 있고, 관리자는 누가 무엇을 할 수 있는지 꼼꼼하게 관리할 수 있다. CVCS를 관리하는 것은 수많은 클라이언트의 로컬 데이터베이스를 관리하는 것보다 훨씬 쉽다.

그러나 CVCS는 심각한 단점이 있다. 중앙 서버가 잘못되면 모든 것이 잘못된다는 점이다. 서버가 다운될 경우 서버가 다시 복구될 때까지 다른 사람과의 협업도, 진행 중이던 작업을 버전 관리하는 것도 불가능해진다. 중앙 데이터베이스가 저장된 하드디스크에 오류가 발생하고 백업도 없다면, 사람들이 각자 자신의 컴퓨터에 가지고 있던 스냅샷 외에는 그동안 쌓인 프로젝트의 이력을 모두 잃게 된다. 로컬 VCS 시스템도 같은 문제가 있다. 프로젝트의 모든 이력이 한곳에만 있을 경우 이것은 피할 수 없는 문제다.

1.1.3 분산 버전 관리 시스템

분산 버전 관리 시스템(Distributed Version Control System; DVCS)은 앞서 말한 문제를 해결하기 위해 개발되었다. Git, Mecurial, Bazaar, Darcs 등 DVCS에서는 클라이언트가 파일들의 마지막 스냅샷을 가져오는 대신 저장소(repository)를 통째로 복제한다. 따라서 서버에 문제가 생겨도 어느 클라이언트든 복제된 저장소를 다시 서버로 복사하면 서버가 복구된다. 체크아웃(checkout)을 할 때마다 전체 백업이 일어나는 셈이다(그림 1-3 참조).

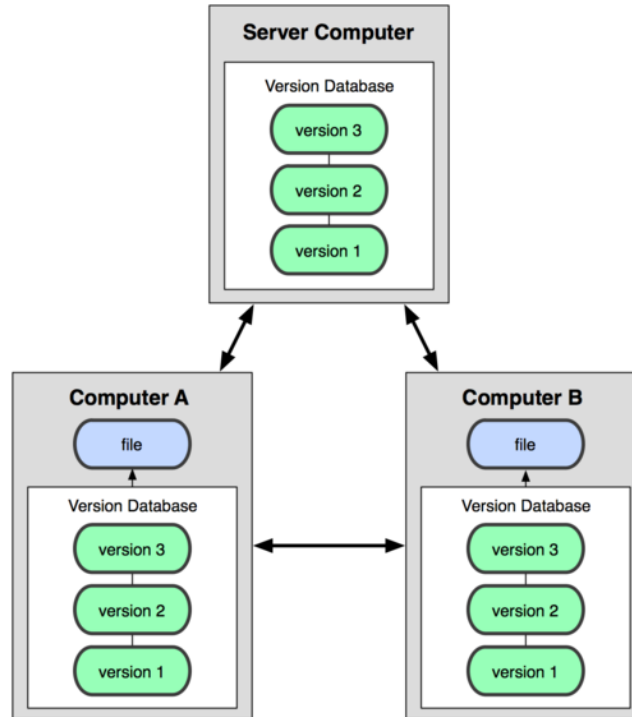



그림 1.3: 분산 버전 관리 시스템 다이어그램

게다가 대부분의 DVCS에서는 다수의 원격 저장소(remote repository)를 갖는 것이 가능하기 때문에 동시에 여러 그룹과 여러 방법으로 함께 작업할 수 있다. 이로 인해 계층 모델(hierarchical model) 등 중앙집중 시스템에서는 할 수 없는 다양한 작업 방식(workflow)들을 사용해볼 수 있다.

1.2 짧게 보는 Git의 역사

인생을 살다 보면 여러 가지 일들이 벌어지듯이 Git의 삶 또한 창조적인 파괴와 모순 속에서 시작되었다. 리눅스 커널은 굉장히 규모가 큰 오픈소스 프로젝트다. 리눅스 커널의 일생에서 대부분 시절은 패치와 단순 압축 파일로만 관리했다. 2002년에 드디어 리눅스 커널은 BitKeeper라고 불리는 상용 DVCS를 사용하기 시작했다.

2005년에 커뮤니티가 만드는 리눅스 커널과 이익을 추구하는 회사가 개발한 BitKeeper의 관계는 틀어졌다. BitKeeper의 무료 사용이 제고된 것이다. 이 사건은 리눅스 개발 커뮤니티(특히 리눅스 창시자 리누스 토발즈)가 자체 도구를 만드는 계기가 됐다. Git은 BitKeeper를 사용하면서 배운 교훈을 기초로 아래와 같은 목표를 세웠다:

- 빠른 속도
- 단순한 구조
- 비선형적인 개발(수천 개의 동시 다발적인 브랜치)
- 완벽한 분산

- 리눅스 커널 같은 대형 프로젝트에도 유용할 것(속도나 데이터 크기 면에서)

Git은 2005년 탄생하고 나서 아직도 초기 목표를 그대로 유지하고 있다. 그러면서도 사용하기 쉽게 진화하고 성숙했다. Git은 미친 듯이 빨라서 대형 프로젝트에 사용하기도 좋다. Git은 동시다발적인 브랜치에도 끄떡없는 슈퍼 울트라 브랜칭 시스템이다(3장 참고).

1.3 Git 기초

Git의 핵심은 뭘까? 이 질문은 Git을 이해하는데 굉장히 중요하다. Git이 무엇이고 어떻게 동작하는지 이해한다면 쉽게 Git을 효과적으로 사용할 수 있다. Git을 배우려면 Subversion이나 Perforce 같은 다른 VCS를 사용하던 경험을 지워버려야 한다. Git은 미묘하게 달라서 다른 VCS에서 쓰던 개념으로는 헷갈릴 거다. 사용자 인터페이스는 매우 비슷하지만, 정보를 취급하는 방식이 다르다. 이런 차이점을 이해하면 Git을 사용하는 것이 어렵지 않다.

1.3.1 델타가 아니라 스냅샷

Subversion과 Subversion 비슷한 놈들과 Git의 가장 큰 차이점은 데이터를 다루는 방법에 있다. 큰 틀에서 봤을 때 대부분의 VCS 시스템이 관리하는 정보는 파일들의 목록이다. CVS, Subversion, Perforce, Bazaar 등의 시스템은 파일의 집합으로 정보를 관리한다. 각 파일의 변화를 그림 1-4처럼 시간순으로 관리한다.

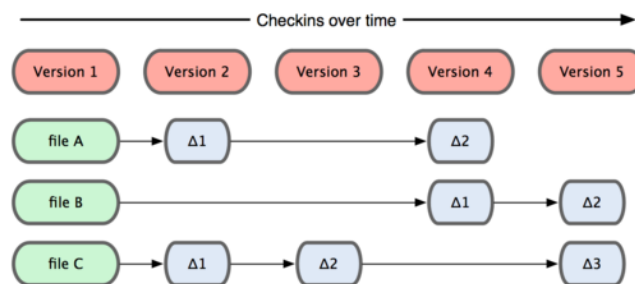



그림 1.4: 각 파일에 대한 변화(델타)를 저장하는 시스템들

Git은 이런 식으로 데이터를 저장하지도 취급하지도 않는다. 대신 Git의 데이터는 파일 시스템의 스냅샷이라 할 수 있으며 크기가 아주 작다. Git은 커밋하거나 프로젝트의 상태를 저장할 때마다 파일이 존재하는 그 순간을 중요하게 여긴다. 파일이 달라지지 않았으면 Git은 성능을 위해서 파일을 저장하지 않는다. 단지 이전 상태의 파일에 대한 링크만 저장한다. Git은 그림 1-5처럼 동작한다.

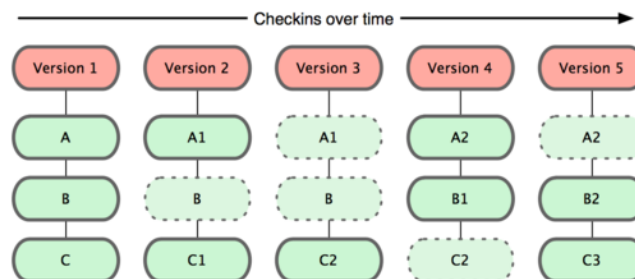



그림 1.5: Git은 시간순으로 프로젝트의 스냅샷을 저장한다

이것이 Git이 다른 VCS와 구분되는 점이다. 이점 때문에 Git는 다른 시스템들이 과거로부터 답습해왔던 버전 컨트롤의 개념과 다르다는 것이고 많은 부분을 새로운 관점에서 바라본다. Git은 강력한 도구

를 지원하는 작은 파일시스템이다. Git은 단순한 VCS가 아니다. 이제3장에서 설명할 Git 브랜치를 사용하면 얻게 되는 이득이 무엇인지 설명한다.

1.3.2 거의 모든 명령을 로컬에서 실행

거의 모든 명령이 로컬 파일과 데이터만 사용하기 때문에 네트워크에 있는 다른 컴퓨터는 필요 없다. 대부분의 명령어가 네트워크의 속도에 영향을 받는 CVCS에 익숙하다면 Git이 매우 놀라울 것이다. Git의 이런 특징에서 나오는 미칠듯한 속도는 오직 Git느님만이 구사할 수 있는 초인적인 능력이다. 프로젝트의 모든 히스토리가 로컬 디스크에 있기 때문에 모든 명령을 순식간에 실행된다.
예를 들어 Git은 프로젝트의 히스토리를 조회할 때 서버 없이 조회한다. 그냥 로컬 데이터베이스에서 히스토리를 읽어서 보여 준다. 그래서 눈 깜짝할 사이에 히스토리를 조회할 수 있다. 어떤 파일의 현재 버전과 한 달 전의 상태를 비교해보고 싶을 때도 Git은 그냥 한 달 전의 파일과 지금의 파일을 로컬에서 찾는다. 파일을 비교하기 위해 리모트에 있는 서버에 접근하고 나서 예전 버전을 가져올 필요가 없다.
즉 오프라인 상태에서도 비교할 수 있다. 비행기나 기차 등에서 작업하고 네트워크에 접속하고 있지 않아도 커밋할 수 있다. 다른 VCS 시스템에서는 불가능한 일이다. Perforce는 서버에 연결할 수 없을 때 할 수 있는 일이 별로 없다. Subversion이나 CVS에서도 마찬가지다. 데이터베이스에 접근할 수 없어서 파일을 편집할 수는 있지만, 커밋할 수 없다. 매우 사소해 보이지만 실제로 이 상황에 부닥쳐보면 느껴지는 차이가 매우 크다.

1.3.3 Git의 무결성

Git은 모든 데이터를 저장하기 전에 체크섬(또는 해시)을 구하고 그 체크섬으로 데이터를 관리한다. 체크섬 없이 어떠한 파일이나 디렉토리도 변경할 수 없다. 체크섬은 Git에서 사용하는 가장 기본적인(Atomic) 데이터 단위이자 Git의 기본 철학이다. Git 없이는 체크섬을 다룰 수 없어서 파일의 상태도 알 수 없고 심지어 데이터를 잃어버릴 수도 없다.
Git은 SHA-1 해시를 사용하여 체크섬을 만든다. 만든 체크섬은 40자 길이의 16진수 문자열이다. 파일의 내용이나 디렉토리 구조를 이용하여 체크섬을 구한다. SHA-1은 아래처럼 생겼다:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Git은 모든 것을 해시로 식별하기 때문에 이런 값은 여기저기서 보인다. 실제로 Git은 파일을 이름으로 저장하지 않고 해당 파일의 해시로 저장한다.

1.3.4 Git은 데이터를 추가할 뿐

Git으로 무얼 하든 데이터를 추가한다. 되돌리거나 데이터를 삭제할 방법이 없다. 다른 VCS처럼 Git도 커밋하지 않으면 변경사항을 잃어버릴 수 있다. 하지만, 일단 스냅샷을 커밋하고 나면 데이터를 잃어버리기 어렵다.
Git을 사용하면 프로젝트가 심각하게 망가질 걱정 없이 매우 즐겁게 여러 가지 실험을 해 볼 수 있다. 9장을 보면 Git이 데이터를 어떻게 저장하고 손실을 어떻게 복구해야 할지 알 수 있다.

1.3.5 세 가지 상태

이 부분은 중요하기에 집중해서 읽어야 한다. Git을 공부하기 위해 반드시 짚고 넘어가야 할 부분이다. Git은 파일을 Committed, Modified, Staged 이렇게 세 가지 상태로 관리한다. Committed란 데이

터가 로컬 데이터베이스에 안전하게 저장됐다는 것을 의미한다. Modified는 수정한 파일을 아직 로컬 데이터베이스에 커밋하지 않은 것을 말한다. Staged란 현재 수정한 파일을 곧 커밋할 것이라고 표시한 상태를 의미한다.

이 세 가지 상태는 Git 프로젝트의 세 가지 단계와 연결돼 있다. Git 디렉토리, 워킹 디렉토리, Staging Area 이렇게 세 가지 단계를 이해하고 넘어가자.

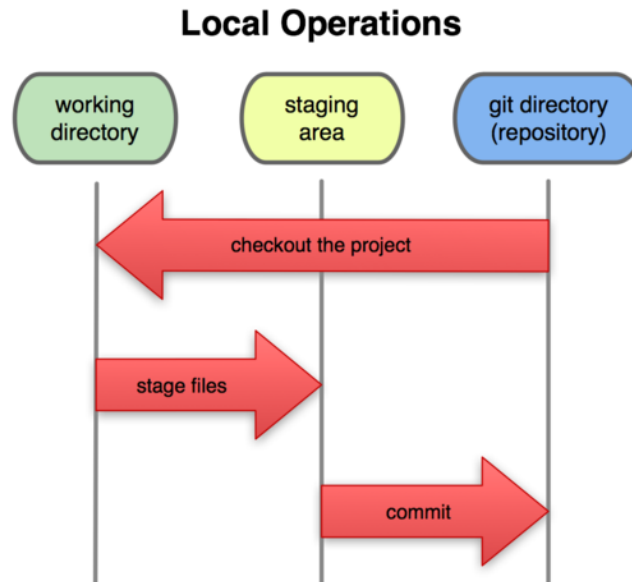



그림 1.6: 워킹 디렉토리, Staging Area, Git 디렉토리

Git 디렉토리는 Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말한다. Git 디렉토리가 Git의 핵심이다. 다른 컴퓨터에 있는 저장소를 Clone 할 때 Git 디렉토리가 만들어진다.

워킹 디렉토리는 프로젝트의 특정 버전을 Checkout한 것이다. Git 디렉토리는 지금 작업하는 디스크에 있고 그 디렉토리에 압축된 데이터베이스에서 파일을 가져와서 워킹 디렉토리를 만든다.

Staging Area는 Git 디렉토리에 있다. 단순한 파일이고 곧 커밋할 파일에 대한 정보를 저장한다. 종종 인덱스라고 불리기도 하지만, Staging Area라는 명칭이 표준이 되어가고 있다.

Git으로 하는 일은 기본적으로 아래와 같다:

- 워킹 디렉토리에서 파일을 수정한다.
- Staging Area에 파일을 Stage해서 커밋할 스냅샷을 만든다.
- Staging Area에 있는 파일들을 커밋해서 Git 디렉토리에 영구적인 스냅샷으로 저장한다.

Git 디렉토리에 있는 파일들은 Committed 상태이다. 파일을 수정하고 Staging Area에 추가했다면 Staged이다. 그리고 Checkout하고 나서 수정했지만, 아직 Staging Area에 추가하지 않았으면 Modified이다. 2장에서 이 상태에 대해 좀 더 자세히 배운다. 특히 Staging Area를 어떻게 이용하는지 혹은 아예 생략하는 방법도 설명한다.

1.4 Git 설치

Git을 사용하려면 우선 설치해야 한다. 다양한 방법으로 Git을 설치할 수 있지만 두 가지 방법이 가장 일반적이다. 하나는 소스코드로 컴파일하여 설치하는 방법이고 다른 하나는 각 운영체제(혹은 플랫폼)의 패키지를 사용하여 설치하는 방법이다.

1.4.1 소스코드로 설치하기

소스코드로 설치하면 Git의 가장 최신 버전을 설치할 수 있기 때문에 컴파일하여 설치할 시간이 있으면 소스코드로 Git을 설치하는 것이 좋다. Git은 계속 UI를 개선하고 있기 때문에 최신 버전을 사용하면 좋은 기능을 빨리 사용할 수 있다. 리눅스 패키지는 보통 최신 버전이 아니고 예전 버전이다. 그래서 Backport를 사용하거나 소스코드로 설치하는 것도 좋은 대안이다.
Git을 설치하려면 아래와 같은 라이브러리들이 필요하다. Git은 curl, zlib, openssl, expat, libiconv를 필요로 한다. 예를 들어 Fedora처럼 yum을 사용하는 시스템이나 apt-get이 있는 데비안류 시스템이면 아래 명령어를 실행하여 의존 패키지를 설치할 수 있다:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
  libz-dev libssl-dev
```

필요한 라이브러리를 모두 설치하고 다음 단계를 진행한다. Git 웹 사이트에서 최신 스냅샷을 가져온다:

```
http://git-scm.com/download
```

그리고 컴파일하고 설치한다:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

설치한 다음부터는 Git을 사용하여 Git 소스코드를 수정할 수 있다:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
```

1.4.2 리눅스에 설치

리눅스에서 패키지로 Git을 설치할 때에는 보통 각 배포판에서 사용하는 패키지 관리도구를 사용하여 설치한다. Fedora에서는 아래와 같이 한다:

```
$ yum install git-core
```

Ubuntu같은 데비안류 배포판에서는 apt-get을 사용한다:

```
$ apt-get install git
```

1.4.3 Mac에 설치하기

Mac에 Git을 쉽게 설치하는 방법은 두 가지가 있다. GUI 인스톨러가 가장 쉽게 사용할 수 있다. SourceForge 페이지에서 내려받는다:

```
http://sourceforge.net/projects/git-osx-installer/
```

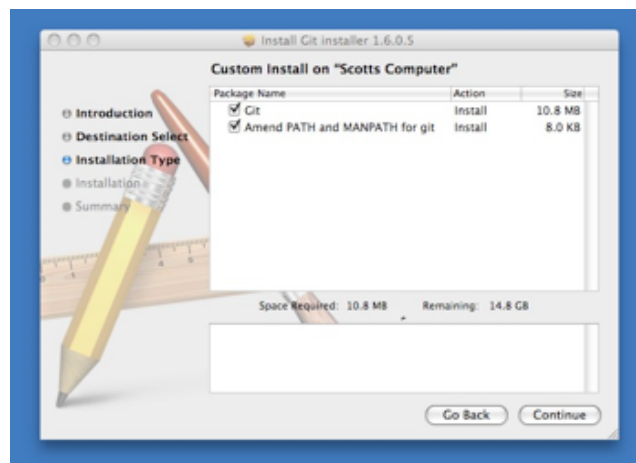



그림 1.7: OS X Git 인스톨러

MacPorts(http://www.macports.org)를 사용하는 방법도 있다. MacPorts가 설치돼 있으면 아래와 같이 Git을 설치한다:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

이제 설치는 했다. 만약 Subversion 저장소를 Git과 함께 사용해야 하면 svn도 필요하다.

1.4.4 윈도에 설치

윈도에서도 Git을 쉽게 설치할 수 있다. 그저 구글 코드 페이지에서 msysGit 인스톨러를 내려받고 실행하면 된다:

```
http://msysgit.github.com/
```

설치가 완료되면 CLI 프로그램과 GUI 프로그램을 둘 다 사용할 수 있다. CLI 프로그램에는 SSH 클라이언트가 포함돼 있기 때문에 유용하다.
Windows 사용자 필독:이 책에서 소개하는 다양한 명령어를 사용하려면 유닉스 스타일의 msysGit 쉘을 사용하는 것이 좋다. 어쩔 수 없이 Windows에 포함된 기본 쉘(Command Prompt, 명령 프롬프트)을 꼭 써야 하면 공백이 포함된 파라미터를 Git 명령어에 넘길 때 작은 따옴표(") 대신 큰 따옴표(" ")를 사용해야 한다. 파라미터 끝에 ⊠ 기호가 있을 때도 큰 따옴표로 파라미터를 감싸야 한다. Windows 쉘에서 ⊠ 기호는 다음 줄로 명령어가 이어짐을 나타낸다.

1.5 Git 최초 설정

Git을 설치하고 나면 Git의 사용 환경을 적절하게 설정해 주어야 한다. 한 번만 설정하면 된다. 설정한 내용은 Git을 업그레이드해도 유지된다. 언제든지 다시 바꿀 수 있는 명령어가 있다.
'git config'라는 도구로 설정 내용을 확인하고 변경할 수 있다. Git은 이 설정에 따라 동작한다. 이때 사용하는 설정 파일은 세 가지나 된다.

- `/etc/gitconfig` 파일: 시스템의 모든 사용자와 모든 저장소에 적용되는 설정이다. `git config --system` 옵션으로 이 파일을 읽고 쓸 수 있다.
- `~/.gitconfig` 파일: 특정 사용자에게만 적용되는 설정이다. `git config --global` 옵션으로 이 파일을 읽고 쓸 수 있다.
- `.git/config`: 이 파일은 Git 디렉토리에 있고 특정 저장소(혹은 현재 작업 중인 프로젝트)에만 적용된다. 각 설정은 역순으로 우선시 된다. 그래서 `.git/config`가 `/etc/gitconfig`보다 우선한다.

윈도용 Git은 `$HOME` 디렉토리(`%USERPROFILE%` 환경변수)에 있는 `.gitconfig` 파일을 찾는다. 보통 `C:\Documents and Settings\$USER` 또는 `C:\Users\$USER` 이다(윈도우에서는 `$USER` 대신 `%USERNAME%`를 사용한다). 그리고 msysGit도 /etc/gitconfig를 가지고 있다. 경로는 MSys 루트에 따른 상대 경로다. 인스톨러로 msysGit을 설치할 때 설치 경로를 선택할 수 있다.

1.5.1 사용자 정보

Git을 설치하고 나서 가장 먼저 해야 하는 것은 사용자 이름과 이메일 주소를 설정하는 것이다. Git은 커밋할 때마다 이 정보를 사용한다. 한 번 커밋한 후에는 정보를 변경할 수 없다:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

다시 말하자면 `--global` 옵션으로 설정한 것은 딱 한 번만 하면 된다. 해당 시스템에서 해당 사용자가 사용할 때에는 이 정보를 사용한다. 만약 프로젝트마다 다른 이름과 이메일 주소를 사용하고 싶으면 `--global` 옵션을 빼고 명령을 실행한다.

1.5.2 편집기

사용자 정보를 설정하고 나면 Git에서 사용할 텍스트 편집기를 고른다. 기본적으로 Git은 시스템의 기본 편집기를 사용하고 보통 Vi나 Vim이다. 하지만, Emacs 같은 다른 텍스트 편집기를 사용할 수 있고 아래와 같이 실행하면 된다:

```
$ git config --global core.editor emacs
```

1.5.3 Diff 도구

Merge 충돌을 해결하기 위해 사용하는 Diff 도구를 설정할 수 있다. vimdiff를 사용하고 싶으면 아래와 같이 실행한다:

```
$ git config --global merge.tool vimdiff
```

이렇게 kdiff3, tkdiff, meld, xxdif, emerge, vimdiff, gvimdiff, ecmerge, opendiff를 사용할 수 있다. 물론 다른 도구도 사용할 수 있다. 자세한 내용은 7장에서 다룬다.

1.5.4 설정 확인

`git config --list` 명령을 실행하면 설정한 모든 것을 보여준다:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Git은 같은 키를 여러 파일(`/etc/gitconfig`와 `~/.gitconfig` 같은)에서 읽기 때문에 같은 키가 여러개 있을 수도 있다. 이러면 Git은 나중 값을 사용한다.
`git config {key}` 명령으로 Git이 특정 Key에 대해 어떤 값을 사용하는지 확인할 수 있다:

```
$ git config user.name
Scott Chacon
```

1.6 도움말 보기

명령어에 대한 도움말이 필요할 때 도움말을 보는 방법은 세 가지다:

```
$ git help <verb>
$ git <verb> --help
$ man git-<verb>
```

예를 들어 아래와 같이 실행하면 config 명령에 대한 도움말을 볼 수 있다:

```
$ git help config
```

도움말은 언제 어디서나 볼 수 있다. 오프라인으로도 볼 수 있다. 도움말과 이 책으로 부족하면 다른 사람의 도움을 받는 것이 필요하다. Freenode IRC 서버(irc.freenode.net)에 있는 `#git`이나 `#github` 채널로 찾아가라. 이 채널에는 보통 수백 명의 사람이 접속해 있다. 이 사람들은 모두 Git에 대해 잘 알고 있다. 기꺼이 도와줄 것이다.

1.7 요약

우리는 Git이 무엇이고 지금까지 사용해 온 다른 CVCS와 어떻게 다른지 배웠다. 시스템에 Git을 설치하고 사용자 정보도 설정했다. 다음 장에서는 Git의 사용법을 배운다.

2장

Git의 기초

Git을 사용하는 방법을 알고 싶은데 한 챕터밖에 읽을 시간이 없다면2장을 읽어야 한다. Git에서 자주 사용하는 명령어는 모두2장에 등장한다.2장을 다 읽으면 저장소를 만들고 설정하는 방법, 파일을 추적하거나(Track) 추적을 그만두는 방법, 변경 내용을 Stage하고 커밋하는 방법을 알게 된다. 그리고 또 파일이나 파일 패턴을 무시하도록 Git을 설정하는 방법, 실수를 쉽고 빠르게 만회하는 방법, 프로젝트 히스토리를 조회하고 커밋을 비교하는 방법, 리모트 저장소에 Push하고 Pull하는 방법을 살펴본다.

2.1 Git 저장소 만들기

Git 저장소를 만드는 방법은 두 가지다. 기존 프로젝트를 Git 저장소로 만드는 방법이 있고 다른 서버에 있는 저장소를 Clone하는 방법이 있다.

2.1.1 기존 디렉토리를 Git 저장소로 만들기

기존 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트의 디렉토리로 이동해서 아래과 같은 명령을 실행한다.

```
$ git init
```

이 명령은 `.git`이라는 하위 디렉토리를 만든다. `.git` 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다(`.git` 디렉토리가 막 만들어진 직후에 어떤 파일이 있는지에 대한 내용은 9장에서 다룬다). 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다.
Git이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다. `git add` 명령으로 파일을 추가하고 커밋한다:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

매우 짧은 시간에 명령어를 몇개 실행해서 Git 저장소를 만들고 파일이 관리되게 했다.

2.1.2 기존 저장소를 Clone하기

다른 프로젝트에 참여하거나(Contribute) Git 저장소를 복사하고 싶을 때 `git clone` 명령을 사용한다. 이미 Subversion 같은 VCS에 익숙한 사용자에게는 `checkout`이 아니라 `clone`이라는 점이 도드라져 보일 것이다. Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 모든 데이터를 복사한다는 것이다. `git clone`을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다(서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다 - 4장에서 좀 더 자세히 다룬다).
`git clone [url]` 명령으로 저장소를 Clone한다. Ruby용 Git 라이브러리인 Grit을 Clone하려면 아래과 같이 실행한다:

```
$ git clone git://github.com/schacon/grit.git
```

이 명령은 "grit"이라는 디렉토리를 만들고 그 안에 `.git` 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout해 놓는다. `grit` 디렉토리로 이동하면 Checkout으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다. 아래과 같은 명령을 사용하여 저장소를 Clone하면 "grit"이 아니라 다른 디렉토리 이름으로 Clone할 수 있다:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

디렉토리 이름이 `mygrit`이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같다.
Git은 다양한 프로토콜을 지원한다. 이제까지는 `git://` 프로토콜을 사용했지만 `http(s)://`를 사용할 수도 있고 `user@server:/path.git`처럼 SSH 프로토콜을 사용할 수도 있다. 자세한 내용은 4장에서 다룬다. 4장에서는 각 프로토콜의 장단점과 Git 저장소에 접근하는 방법을 설명한다.

2.2 수정하고 저장소에 저장하기

만질 수 있는 Git 저장소를 하나 만들었고 워킹 디렉토리에 Checkout도 했다. 이제는 파일을 수정하고 파일의 스냅샷을 커밋해 보자. 파일을 수정하다가 저장하고 싶으면 스냅샷을 커밋한다.
워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나눈다. Tracked 파일은 이미 스냅샷에 포함돼 있던 파일이다. Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋하면 저장소에 기록되는) 상태 중 하나이다. 그리고 나머지 파일은 모두 Untracked 파일이다. Untracked 파일은 워킹 디렉토리에 있는 모든 파일이 스냅샷에 포함돼 있는 것은 아니고 Staging Area에 있는 것도 아니다. 처음 저장소를 Clone하면 모든 파일은 Tracked이면서 Unmodified 상태가 된다. 파일을 Checkout하고 나서 아무것도 수정하지 않았기 때문에 그렇다.
마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일이 수정되면 Git은 그 즉시 파일을 Modified 상태로 인식한다. 그리고 이 수정한 파일을 Stage하고 Staged 상태인 파일을 커밋한다. 이 라이프사이클을 그림 2-1처럼 계속 반복한다.

2.2.1 파일의 상태 확인하기

파일의 상태를 확인하려면 보통 `git status` 명령을 사용한다. Clone한 후에 바로 이 명령을 실행하면 아래과 같은 메시지를 볼 수 있다:

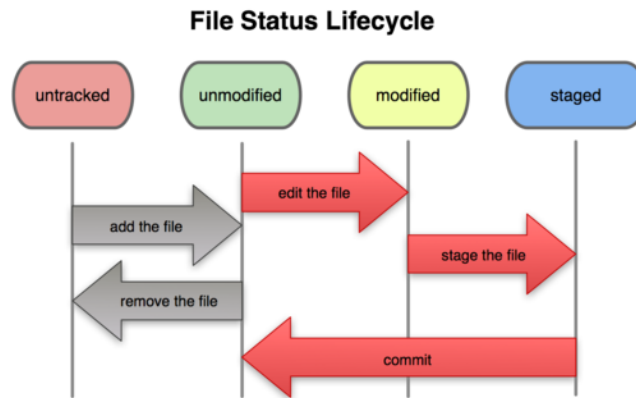



그림 2.1: 파일의 라이프사이클

```
$ git status
On branch master
nothing to commit, working directory clean
```

위의 내용은 파일을 하나도 수정하지 않았다는 것을 말해준다. Tracked나 Modified 상태인 파일이 없다는 의미다. Untracked 파일은 아직 없어서 목록에 나타나지 않는다. 그리고 현재 작업 중인 브랜치를 알려준다. 기본 브랜치가 master이기 때문에 현재 master로 나오는 것이다. 브랜치 관련 내용은 차차 알아가자. 다음 장에서 브랜치와 레퍼런스에 대해 자세히 다룬다.

프로젝트에 `README` 파일을 만들어보자. `README` 파일은 새로 만든 파일이기 때문에 `git status`를 실행하면 'Untracked files'에 들어 있다:

```
$ vim README
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

README

nothing added to commit but untracked files present (use "git add" to track)
```

`README` 파일은 `Untracked files` 부분에 속해 있는데 이것은 `README` 파일이 Untracked 상태라는 것을 말한다. Git은 Untracked 파일을 아직 스냅샷(커밋)에 넣어지지 않은 파일이라고 본다. 파일이 Tracked 상태가 되기 전까지는 Git은 절대 그 파일을 커밋하지 않는다. 그래서 일하면서 생성하는 바이너리 파일 같은 것을 커밋하는 실수는 하지 않게 된다. README 파일을 추가해서 직접 Tracked 상태로 만들어 보자.

2.2.2 파일을 새로 추적하기

`git add` 명령으로 파일을 새로 추적할 수 있다. 아래 명령을 실행하면 Git은 `README` 파일을 추적한다:

```
$ git add README
```

git status 명령을 다시 실행하면 README 파일이 Tracked 상태이면서 Staged 상태라는 것을 확인할 수 있다:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README
```

'Changes to be committed' 에 들어 있는 파일은 Staged 상태라는 것을 의미한다. 커밋하면 git add를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다. 앞에서 git init 명령을 실행했을 때, 그 다음 git add (files) 명령을 실행했던 걸 기억할 것이다. 이것은 작업 디렉토리에 있는 파일들을 추적하기 시작하게 하였다. git add 명령은 파일 또는 디렉토리의 경로명을 아규먼트로 받는다; 만일 디렉토리를 아규먼트로 줄 경우, 그 디렉토리 아래에 있는 모든 파일들을 재귀적으로 추가한다.

2.2.3 Modified 상태의 파일을 Stage하기

이미 Tracked 상태인 파일을 수정하는 법을 알아보자. benchmarks.rb라는 파일을 수정하고 나서 git status 명령을 다시 실행하면 결과는 아래와 같다:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

이 benchmarks.rb 파일은 Changes not staged for commit에 있다. 이것은 수정한 파일이 Tracked 상태이지만 아직 Staged 상태는 아니라는 것이다. Staged 상태로 만들려면 git add 명령을 실행해야 한다. git add는 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용한다. git add를 실행하여 benchmarks.rb 파일을 Staged 상태로 만들고 git status 명령으로 결과를 확인해보자:

```
$ git add benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README
        modified:   benchmarks.rb
```

두 파일 모두 Staged 상태이므로 다음 커밋에 포함된다. 하지만, 아직 더 수정해야 한다는 것을 알게 되어 바로 커밋하지 못하는 상황이 되었다고 하자. 이 상황에서 benchmark.rb 파일을 열고 수정한다. 아마 당신은 커밋할 준비가 다 됐다고 생각할 테지만, Git은 그렇지 않다. `git status` 명령으로 파일의 상태를 다시 확인해보자:

```
$ vim benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README
        modified:   benchmarks.rb

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)

        modified:   benchmarks.rb
```

헉! benchmarks.rb가 Staged 상태이면서 동시에 Unstaged 상태로 나온다. 어떻게 이런 일이 가능할까? `git add` 명령을 실행하면 Git은 파일을 바로 Staged 상태로 만든다. 지금 이 시점에서 커밋을 하면 `git commit` 명령을 실행하는 시점의 버전이 커밋되는 것이 아니라 마지막으로 `git add` 명령을 실행했을 때의 버전이 커밋된다. 그러니까 `git add` 명령을 실행한 후에 또 파일을 수정하면 `git add` 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다:

```
$ git add benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README
```

```
        modified:   benchmarks.rb
```

2.2.4 파일 무시하기

어떤 파일은 Git이 자동으로 추가하거나 Untracked 파일이라고 보여줄 필요가 없다. 보통 로그 파일이나 빌드 시스템이 자동으로 생성한 파일이 그렇다. 그런 파일을 무시하려면 `.gitignore` 파일을 만들고 그 안에 무시할 파일 패턴을 적는다. 아래는 `.gitignore` 파일의 예이다:

```
$ cat .gitignore
*.[oa]
*~
```

첫번째 줄은 확장자가 `.o` 나 `.a`인 파일을 Git이 무시하라는 것이고 둘째 줄은 `~`로 끝나는 모든 파일을 무시하라는 것이다. `.o`와 `.a`는 각각 빌드 시스템이 만들어내는 오브젝트와 아카이브 파일이고 `~`로 끝나는 파일은 Emacs나 VI 같은 텍스트 편집기가 임시로 만들어내는 파일이다. 또 log, tmp, pid 같은 디렉토리나, 자동으로 생성하는 문서 같은 것들도 추가할 수 있다. `.gitignore` 파일은 보통 처음에 만들어 두는 것이 편리하다. 그래서 Git 저장소에 커밋하고 싶지 않은 파일을 실수로 커밋하는 일을 방지할 수 있다.
.gitignore 파일에 입력하는 패턴은 아래 규칙을 따른다:

- 아무것도 없는 줄이나, `#`로 시작하는 줄은 무시한다.
- 표준 Glob 패턴을 사용한다.
- 디렉토리는 슬래시(`/`)를 끝에 사용하는 것으로 표현한다.
- 느낌표(`!`)로 시작하는 패턴의 파일은 무시하지 않는다.

Glob 패턴은 정규표현식을 단순하게 만든 것으로 생각하면 되고 보통 쉘에서 많이 사용한다. 애스터리스크(`*`)는 문자가 하나도 없거나 하나 이상을 의미하고, `[abc]`는 중괄호 안에 있는 문자 중 하나를 의미한다(그러니까 이 경우에는 a, b, c). 물음표(`?`)는 문자 하나를 말하고, `[0-9]`처럼 중괄호 안의 캐릭터 사이에 하이픈(`-`)을 사용하면 그 캐릭터 사이에 있는 문자 하나를 말한다.
다음은 .gitignore 파일의 예이다:

```
# a comment - 이 줄은 무시한다.
# 확장자가 .a인 파일 무시
*.a
# 윗 줄에서 확장자가 .a인 파일은 무시하게 했지만 lib.a는 무시하지 않는다.
!lib.a
# 루트 디렉토리에 있는 TODO파일은 무시하고 subdir/TODO처럼 하위디렉토리에 있는 파일은 무시하
지 않는다.
/TODO
# build/ 디렉토리에 있는 모든 파일은 무시한다.
build/
# `doc/notes.txt`같은 파일은 무시하고 doc/server/arch.txt같은 파일은 무시하지 않는다.
doc/*.txt
```

```
# `doc` 디렉토리 아래의 모든 .txt 파일을 무시한다.
doc/**/*.txt
```

`**/` 스타일의 문법은 Git 1.8.2 버전부터 사용할 수 있다.

2.2.5 Staged와 Unstaged 상태의 변경 내용을 보기

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보기엔 `git status` 명령이 아니라 `git diff` 명령을 사용해야 한다. 보통우리는 '수정했지만, 아직 Staged 파일이 아닌것?'과 '어떤 파일이 Staged 상태인지?'가 궁금하기 때문에 `git status` 명령으로도 충분하다. `git diff`는 Patch처럼 어떤 라인을 추가했고 삭제했는지가 궁금할 때에 사용한다. `git diff`는 나중에 더 자세히 다룬다.

README 파일을 수정해서 Staged 상태로 만들고 benchmarks.rb 파일은 그냥 수정만 해둔다. 이 상태에서 `git status` 명령을 실행하면 아래와 같은 메시지를 볼 수 있다:

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

`git diff` 명령을 실행하면 수정했지만 아직 staged 상태가 아닌 파일을 비교해 볼 수 있다:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
           @commit.parents[0].parents[0].parents[0]
         end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
```

```
        run_code(x, 'commits 2') do
          log = git.commits('master', 15)
          log.size
```

이 명령은 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage하지 않은 것을 보여준다.

만약 커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶으면 `git diff --cached` 옵션을 사용한다(Git 버전 1.6.1부터는 좀 더 기억하기 쉽게 `git diff --staged`로도 사용할 수 있다). 이 명령은 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

꼭 잊지 말아야 할 것이 있는데 `git diff` 명령은 마지막으로 커밋한 후에 수정한 것들 전부를 보여주지 않는다. `git diff`는 Unstaged 상태인 것들만 보여준다. 이 부분이 조금 헷갈릴 수 있다. 수정한 파일을 모두 Staging Area에 넣었다면 `git diff` 명령은 아무것도 출력하지 않는다.

benchmarks.rb 파일을 Stage한 후에 다시 수정해도 `git diff` 명령을 사용할 수 있다. 이때는 Staged 상태인 것과 Unstaged 상태인 것을 비교한다:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   benchmarks.rb

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
```

```
        modified:   benchmarks.rb
```

`git diff` 명령으로 Unstaged 상태인 변경 부분을 확인해 볼 수 있다:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
 main()

 ##pp Grit::GitRuby.cache_client.stats
+# test line
```

Staged 상태인 파일은 `git diff --cached` 옵션으로 확인한다:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
          @commit.parents[0].parents[0].parents[0]
        end

+        run_code(x, 'commits 1') do
+          git.commits.size
+        end
+
        run_code(x, 'commits 2') do
          log = git.commits('master', 15)
          log.size
```

2.2.6 변경사항 커밋하기

수정한 것을 커밋하기 위해 Staging Area에 파일을 정리했다. Unstaged 상태의 파일은 커밋되지 않는다는 것을 기억해야 한다. Git은 생성하거나 수정하고 나서 `git add` 명령으로 추가하지 않은 파일은 커밋하지 않는다. 그 파일은 여전히 Modified 상태로 남아 있다. 커밋하기 전에 `git status` 명령으로 모든 것이 Staged 상태인지 확인할 수 있다. 그리고 `git commit`을 실행하여 커밋한다:

```
$ git commit
```

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다(아래 예제는 Vim 편집기의 화면이다). 이 편집기는 쉘의 $EDITOR 환경 변수에 등록된 편집기이고 보통은 Vim이나 Emacs을 사용한다. 또 1장에서 설명했듯이 `git config --global core.editor` 명령으로 어떤 편집기를 사용할지 설정할 수 있다:
편집기는 아래와 같은 내용을 표시한다(아래 예제는 Vim 편집기):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#       new file:   README
#       modified:   benchmarks.rb
#
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C
```

자동으로 생성되는 커밋 메시지의 첫 줄은 비어 있고 둘째 줄부터 `git status` 명령의 결과가 채워진다. 커밋한 내용을 쉽게 기억할 수 있도록 이 메시지를 포함할 수도 있고 메시지를 전부 지우고 새로 작성할 수 있다(수정한 내용을 좀 더 구체적으로 남겨 둘 수 있다. `git commit`에 -v 옵션을 추가하면 편집기에 diff 메시지도 추가된다).
메시지를 인라인으로 첨부할 수도 있다. `commit` 명령을 실행할 때 아래와 같이 `-m` 옵션을 사용한다:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master 463dc4f] Story 182: Fix benchmarks for speed
 2 files changed, 3 insertions(+)
 create mode 100644 README
```

`commit` 명령은 몇 가지 정보를 출력하는데 위 예제는 master 브랜치에 커밋했고 체크섬은 `463dc4f`이라고 알려준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 줄이 몇 줄인지 알려준다.
Git은 Staging Area에 속한 스냅샷을 커밋한다는 것을 기억해야 한다. 수정은 했지만, 아직 Staging Area에 넣지 않은 것은 다음에 커밋할 수 있다. 커밋할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

2.2.7 Staging Area 생략하기

Staging Area는 커밋할 파일을 정리한다는 점에서 매우 유용하지만 복잡하기만 하고 필요하지 않은 때도 있다. 아주 쉽게 Staging Area를 생략할 수 있다. `git commit` 명령을 실행할 때 `-a` 옵션을 추가하

면 Git은 Tracked 상태의 파일을 자동으로 Staging Area에 넣는다. 그래서 `git add` 명령을 실행하는 수고를 덜 수 있다:

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb

no changes added to commit (use "git add" and/or "git commit -a")
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+)
```

이 예제에서는 커밋하기 전에 `git add` 명령으로 benchmarks.rb 파일을 추가하지 않았다는 점을 눈여겨보자.

2.2.8 파일을 삭제하기

Git에서 파일을 제거하려면 `git rm` 명령으로 Tracked 상태의 파일을 삭제한 후에(정확하게는 Staging Area에서 삭제하는 것) 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 지워진다.

만약 Git없이 그냥 파일을 삭제하고 `git status` 명령으로 상태를 확인하면 `Changes not staged for commit`(즉, Unstaged) 에 속한다는 것을 확인할 수 있다:

```
$ rm grit.gemspec
$ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        deleted:    grit.gemspec

no changes added to commit (use "git add" and/or "git commit -a")
```

그리고 `git rm` 명령을 실행하면 삭제한 파일은 staged 상태가 된다:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
```

```
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    grit.gemspec
```

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다. 이미 파일을 수정했거나 Index에 (역주, Staging Area을 Git Index라고도 부른다) 추가했다면 `-f`옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 한 번도 커밋한적 없는 데이터는 Git으로 복구할 수 없다.

또 Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다. 이것은 `.gitignore` 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 `.a` 파일 같은 것을 실수로 추가했을 때 쓴다. `--cached` 옵션을 사용하여 명령을 실행한다:

```
$ git rm --cached readme.txt
```

여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 `git rm` 명령에 file-glob 패턴을 사용한다:

```
$ git rm log/\*.log
```

`*`앞에 `\`을 사용한 것을 기억하자. 파일명 확장 기능은 쉘에만 있는 것이 아니라 Git 자체에도 있기 때문에 필요하다. Windows 기본 쉘을 쓸 때는 `\` 기호를 붙이지 않는다. 이 명령은 `log/` 디렉토리에 있는 `.log` 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다:

```
$ git rm \*~
```

이 명령은 `~`로 끝나는 파일을 모두 삭제한다.

2.2.9 파일 이름 변경하기

Git은 다른 VCS 시스템과는 달리 파일 이름의 변경이나 파일의 이동을 명시적으로 관리하지 않는다. 다시 말해서 파일 이름이 변경됐다는 별도의 정보를 저장하지 않는다. Git은 똑똑해서 굳이 파일 이름이 변경되었다는 것을 추적하지 않아도 아는 방법이 있다. 파일의 이름이 변경된 것을 Git이 어떻게 알아내는지 살펴보자.

이렇게 말하고 Git에 `mv` 명령이 있는 게 좀 이상하겠지만, 아래와 같이 파일이름을 변경할 수 있다:

```
$ git mv file_from file_to
```

잘 동작한다. 이 명령을 실행하고 Git의 상태를 확인해보면 Git은 이름이 바뀐 사실을 알고 있다:

```
$ git mv README.txt README
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        renamed:    README.txt -> README
```

사실 `git mv` 명령은 아래 명령어들을 수행한 것과 완전히 똑같다:

```
$ mv README.txt README
$ git rm README.txt
$ git add README
```

`git mv`는 일종의 단축 명령어이다. 이 명령으로 파일이름을 바꿔도 되고 `mv` 명령으로 파일이름을 직접 바꿔도 된다. 단지 Git의 `mv`명령은 편리하게 명령을 세 번 실행해주는 것뿐이다. 어떤 도구로 이름을 바꿔도 상관없다. 중요한 것은 이름을 변경하고 나서 꼭 rm/add 명령을 실행해야 한다는 것뿐이다.

2.3 커밋 히스토리 조회하기

새로 저장소를 만들어서 몇 번 커밋을 했을 수도 있고, 커밋 히스토리가 있는 저장소를 Clone했을 수도 있다. 어쨌든 가끔 저장소의 히스토리를 보고 싶을 때가 있다. Git에는 히스토리를 조회하는 명령어인 `git log`가 있다.
이 예제에서는 simplegit이라는 매우 단순한 프로젝트를 사용한다. simplegit은 Git을 설명하는데 자주 사용하는 예제다. 아래와 같이 이 프로젝트를 Clone한다:

```
git clone git://github.com/schacon/simplegit-progit.git
```

이 프로젝트 디렉토리에서 `git log` 명령을 실행하면 아래와 같이 출력된다:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
```

```
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

특별한 아규먼트 없이 `git log` 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

원하는 히스토리를 검색할 수 있도록 `git log` 명령은 매우 다양한 옵션을 지원한다. 여기에서는 자주 사용하는 옵션을 설명한다.

`-p`가 가장 유용한 옵션 중 하나다. `-p`는 각 커밋의 diff 결과를 보여준다. 게다가 `-2`는 최근 두 개의 결과만 보여주는 옵션이다:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,7 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com"
```

```
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

이 옵션은 직접 diff를 실행한 것과 같은 결과를 출력하기 때문에 동료가 무엇을 커밋했는지 리뷰하고 빨리 조회하는데 유용하다.

가끔은 diff 결과를 줄 단위로 보기보다는 단어 단위로 보는 것이 좋을 때도 있다. `git log -p`와 같은 명령에 `--word-diff` 옵션을 사용하면 줄 단위 대신 단어 단위로 변경사항을 보여준다. 단어 단위로 다른 부분을 확인하는 것은 소스코드에는 별로 유용하지 않다. 책이나 에세이 같이 문장이 긴 글을 쓸 때는 단어 단위로 보는 것이 편하다. `--word-diff` 옵션은 다음과 같이 사용한다:

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

위의 예제는 줄 단위로 보여주는 일반적인 diff와 좀 다르다. 줄 안에서 변경한 부분을 단어 단위로 표시한다. 추가한 단어는 {+ +} 기호가 둘러싸고 삭제한 단어는 [- -] 기호가 둘러싼다. diff는 기본적으로 다른 줄과 위아래 줄을 포함해서 3줄을 보여준다. 줄 단위가 아니라 단어 단위로 비교해서 볼 때는 굳이 3줄을 다 볼 필요가 없다. 예제에서 처럼 -U1 옵션을 주면 해당 줄만 보여준다.

또 git log 명령에는 히스토리의 통계를 보여주는 옵션도 있다. --stat 옵션으로 각 커밋의 통계 정보를 조회할 수 있다:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 file changed, 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |    6 ++++++
 Rakefile         |   23 +++++++++++++++++++++++
 lib/simplegit.rb |   25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+)
```

이 결과에서 --stat 옵션은 어떤 파일이 수정됐는지, 얼마나 많은 파일이 변경됐는지, 또 얼마나 많은 줄을 추가하거나 삭제했는지 보여준다. 요약정보는 가장 뒤쪽에 보여준다.

다른 또 유용한 옵션은 --pretty 옵션이다. 이 옵션을 통해 log의 내용을 보여줄 때 기본 형식 이외에 여러 가지 중에 하나를 선택할 수 있다. oneline 옵션은 각 커밋을 한 줄로 보여준다. 이 옵션은 많은 커밋을 한 번에 조회할 때 유용하다. 추가로 short, full, fuller 옵션도 있는데 이것은 정보를 조금씩 가감해서 보여준다:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

가장 재밌는 옵션은 `format` 옵션이다. 나만의 포맷으로 결과를 출력하고 싶을 때 사용한다. 특히 결과를 다른 프로그램으로 파싱하고자 할 때 유용하다. 이 옵션을 사용하면 포맷을 정확하게 일치시킬 수 있기 때문에 Git을 새 버전으로 바꿔도 결과 포맷이 바뀌지 않는다:

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

표 2-1 형식에서 사용하는 유용한 옵션들.

```
Option Description of Output
%H Commit hash
%h Abbreviated commit hash
%T Tree hash
%t Abbreviated tree hash
%P Parent hashes
%p Abbreviated parent hashes
%an Author name
%ae Author e-mail
%ad Author date (format respects the --date= option)
%ar Author date, relative
%cn Committer name
%ce Committer email
%cd Committer date
%cr Committer date, relative
%s Subject
```

저자(Author) 와 커미터(Committer) 를 구분하는 것이 조금 이상해 보일 수 있다. 저자는 원래 작업을 수행한 원작자이고 커밋터는 마지막으로 이 작업을 적용한 사람이다. 만약 당신이 어떤 프로젝트에 패치를 보냈고 그 프로젝트의 담당자가 패치를 적용했다면 두 명의 정보를 모두 알 필요가 있다. 그래서 이 경우 당신이 저자고 그 담당자가 커미터다. 5장에서 이 주제에 대해 자세히 다룰 것이다.

`oneline`과 `format` 옵션은 `--graph` 옵션과 함께 사용할 때 더 빛난다. 이 명령은 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력한다. 이 명령을 Grit 프로젝트 저장소에서 사용해보면 아래와 같다:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

`git log` 명령의 기본적인 옵션과 출력물의 형식에 관련된 옵션을 살펴보았다. `git log` 명령은 앞서 살펴본 것보다 더 많은 옵션을 지원한다. 표 2-2 는 지금 설명한 것과 함께 유용하게 사용할 수 있는 옵션이다. 각 옵션으로 어떻게 `log` 명령을 제어할 수 있는지 보여준다.

```
옵션 설명
-p 각 커밋에 적용된 패치를 보여준다.
--word-diff diff 결과를 단어 단위로 보여준다.
--stat 각 커밋에서 수정된 파일의 통계정보를 보여준다.
--shortstat `--stat` 명령의 결과 중에서 수정한 파일, 추가된 줄, 삭제된 줄만 보여준다.
--name-only 커밋 정보중에서 수정된 파일의 목록만 보여준다.
--name-status 수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.
--abbrev-commit 40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.
--relative-date 정확한 시간을 보여주는 것이 아니라 `2 주전`처럼 상대적인 형식으로 보여준다.
--graph 브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
--pretty 지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.
--oneline `--pretty=oneline --abbrev-commit` 옵션을 함께 사용한 것과 동일하다.
```

2.3.1 조회 제한조건

출력 형식과 관련된 옵션을 살펴봤지만 `git log` 명령은 조회 범위를 제한하는 옵션들도 있다. 히스토리 전부가 아니라 부분만 조회한다. 이미 최근 두 개만 조회하는 `-2` 옵션은 살펴봤다. 실제 사용법은 `-<n>`이고 n은 최근 n개의 커밋을 의미한다. 사실 이 옵션은 잘 쓰이지 않는다. Git은 기본적으로 출력을 pager류의 프로그램을 거쳐서 내보내므로 한 번에 한 페이지씩 보여준다.

반면 `--since`나 `--until`같은 시간을 기준으로 조회하는 옵션은 매우 유용하다. 지난 2주 동안 만들어진 커밋들만 조회하는 명령은 아래와 같다:

```
$ git log --since=2.weeks
```

이 옵션은 다양한 형식을 지원한다. `2008-01-15`같이 정확한 날짜도 사용할 수 있고 `2 years 1 day 3 minutes ago`같이 상대적인 기간을 사용할 수도 있다.
또 다른 기준도 있다. `--author` 옵션으로 저자를 지정하여 검색할 수도 있고 `--grep` 옵션으로 커밋 메시지에서 키워드를 검색할 수도 있다(author와 grep 옵션을 함께 사용하면 모두 만족하는 커밋을 찾는다).
grep 옵션을 여러개 사용하면 그중 하나이상 만족하는 커밋을 찾는다. 모두 만족하는 커밋을 찾으려면 `--all-match` 옵션을 추가해야 한다.
마지막으로 파일 경로로 검색하는 옵션이 있는데 이것도 정말 유용하다. 디렉토리나 파일 이름을 사용하여 그 파일이 변경된 log의 결과를 검색할 수 있다. 이 옵션은 --와 함께 경로 이름을 사용하는데 명령어 끝 부분에 쓴다(역주, `git log -- path1 path2`).
표 2-3은 조회 범위를 제한하는 옵션들이다.

```
옵션 설명
-(n) 최근 n 개의 커밋만 조회한다.
--since, --after 명시한 날짜 이후의 커밋만 검색한다.
--until, --before 명시한 날짜 이전의 커밋만 조회한다.
--author 입력한 저자의 커밋만 보여준다.
--committer 입력한 커미터의 커밋만 보여준다.
```

아래 예제는 2008년 10월에 Junio Hamano가 커밋한 히스토리를 조회하는 것이다. 그 중에서 테스트 파일을 수정한 커밋 중에서 머지 커밋이 아닌 것들만 조회한다:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

총 2만여 개의 커밋 히스토리에서 이 명령의 검색 조건에 만족하는 것은 단 6개였다.

2.3.2 GUI 도구로 히스토리를 시각화하기

GUI 도구로 커밋 히스토리를 시각화하고 싶다면 gitk를 사용할 수 있다. gitk는 Tcl/Tk 프로그램이고 `git log` 명령을 시각화해주는 도구다. gitk는 `git log` 명령이 지원하는 필터링 옵션을 거의 모두 지원한다. 프로젝트 디렉토리에서 gitk를 실행하면 그림 2-2처럼 보일 것이다.
위쪽 반을 차지하는 윈도에서는 히스토리를 그래프로 예쁘게 보여준다. 아래쪽 반을 차지하는 윈도는 diff 결과를 보여주는데 위쪽 윈도에서 선택한 커밋에 대한 diff 결과를 보여준다.

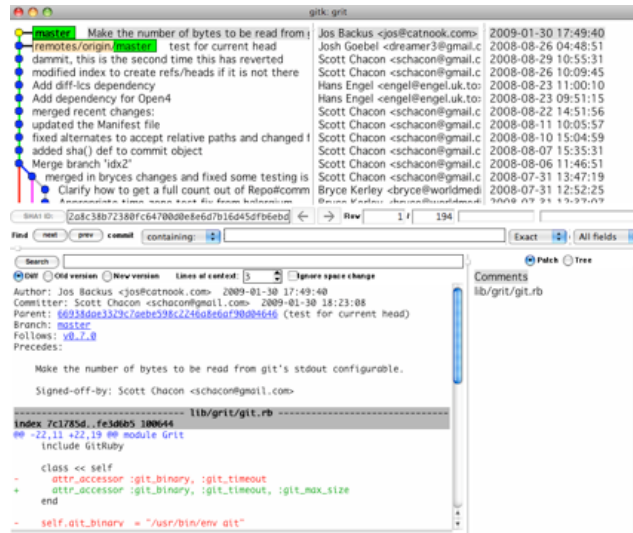



그림 2.2: gitk의 히스토리

2.4 되돌리기

일을 하다보면 모든 단계에서 어떤 것은 되돌리고(Undo) 싶을 때가 있다. 이번에는 우리가 한 일을 되돌리는 방법을 살펴볼 것이다. 한 번 되돌리면 복구할 수 없어서 주의해야 한다. Git을 사용하면 우리가 한 실수를 복구하지 못할 것은 거의 없지만 되돌리기는 복구할 수 없다.

2.4.1 커밋 수정하기

종종 완료한 커밋을 수정해야 할 때가 있다. 너무 일찍 커밋했거나 어떤 파일을 빼먹었을 때 그리고 커밋 메시지를 잘못 적었을 때 하게 된다. 다시 커밋하고 싶으면 --amend 옵션을 사용한다:

```
$ git commit --amend
```

이 명령은 Staging Area를 사용하여 커밋한다. 만약 마지막으로 커밋하고 나서 수정한 것이 없다면(커밋하자마자 바로 이 명령을 실행하는 경우) 조금 전에 한 커밋과 모든 것이 같다. 이때는 커밋 메시지만 수정한다.

편집기가 실행되면 이전 커밋 메시지가 자동으로 포함된다. 메시지를 수정하지 않고 그대로 커밋해도 기존의 커밋을 덮어쓴다.

커밋을 했는데 Stage하는 것을 깜빡하고 빠트린 파일이 있으면 아래와 같이 고칠 수 있다:

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

여기서 실행한 명령어 3개는 모두 하나의 커밋으로 기록된다. 두 번째 커밋은 첫 번째 커밋을 덮어쓴다.

2.4.2 파일 상태를 Unstage로 변경하기

다음은 Staging Area와 워킹 디렉토리 사이를 넘나드는 방법을 설명한다. 두 영역의 상태를 확인할 때마다 변경된 상태를 되돌리는 방법을 알려주기 때문에 매우 편리하다. 예를 들어 파일을 두 개 수정하고서 따로따로 커밋하려고 했지만, 실수로 `git add *` 라고 실행해 버렸다. 두 파일 모두 Staging Area에 들어 있다. 이제 둘 중 하나를 어떻게 꺼낼까? 우선 `git status` 명령으로 확인해보자:

```
$ git add .
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
        modified:   benchmarks.rb
```

`Changes to be commited` 밑에 `git reset HEAD <file>...` 이라는 문장을 볼 수 있다. 이 명령으로 Unstage 상태로 변경할 수 있다. benchmarks.rb 파일을 Unstage 상태로 변경해보자:

```
$ git reset HEAD benchmarks.rb
Unstaged changes after reset:
M       benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

명령어가 낯설게 느껴질 수도 있지만 잘 동작한다. benchmarks.rb 파일은 Unstage 상태가 됐다.

2.4.3 Modified 파일 되돌리기

어떻게 해야 benchmarks.rb 파일을 수정하고 나서 다시 되돌릴 수 있을까? 그러니까 최근 커밋된 버전으로(아니면 처음 Clone했을 때처럼 워킹 디렉토리에 처음 Checkout 한 그 내용으로) 되돌리는 방법이 무얼까? `git status` 명령이 친절하게 알려준다. 바로 위에 있는 예제에서 Unstaged 부분을 보자:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   benchmarks.rb
```

위의 메시지는 수정한 파일을 되돌리는 방법을 꽤 정확하게 알려준다(적어도 Git 1.6.1이후 버전부터는 그렇다. 만약 예전 것을 아직 사용하고 있으면 업그레드하는 것이 좋다. 편의성이 많이 개선됐다). 알려주는 대로 한 번 해보자:

```
$ git checkout -- benchmarks.rb
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   README.txt
```

정상적으로 복원된 것을 알 수 있다. 하지만 이 명령은 꽤 위험한 명령이라는 것을 알아야 한다. 수정 이전의 파일로 덮어썼기 때문에 수정했던 내용은 전부 사라진다. 수정한 내용이 진짜 마음에 들지 않을 때에만 사용하자. 정말 이렇게 삭제해야 한다면 Stash와 Branch를 사용하자. 다음 장에서 다루는 이 방법들이 훨씬 낫다.

Git으로 커밋한 모든 것은 언제나 복구할 수 있다. 삭제한 브랜치에 있었던 것도 `--amend` 옵션으로 다시 커밋한 것도 복구할 수 있다(자세한 것은 9장에서 다룬다). 하지만, 커밋하지 않고 잃어버린 것은 절대로 되돌릴 수 없다.

2.5 리모트 저장소

리모트 저장소를 관리할 줄 알아야 다른 사람과 함께 일할 수 있다. 리모트 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말한다. 저장소는 여러 개가 있을 수 있는데 어떤 저장소는 읽고 쓰기 모두 할 수 있고 어떤 저장소는 읽기 권한만 있을 수도 있다. 간단히 말해서 다른 사람들과 함께 일한다는 것은 리모트 저장소를 관리하면서 데이터를 거기에 Push하고 Pull하는 것이다. 리모트 저장소를 관리한다는 것은 저장소를 추가, 삭제하는 것뿐만 아니라 브랜치를 관리하고 추적할지 말지 등을 관리하는 것을 말한다. 이번에는 리모트 저장소를 관리하는 방법에 대해 설명한다.

2.5.1 리모트 저장소 확인하기

`git remote` 명령으로 현재 프로젝트에 등록된 리모트 저장소를 확인할 수 있다. 이 명령은 리모트 저장소의 단축 이름을 보여준다. 저장소를 Clone하면 origin이라는 리모트 저장소가 자동으로 등록되기 때문에 origin이라는 이름을 볼 수 있다:

```
$ git clone git://github.com/schacon/ticgit.git
Cloning into 'ticgit'...
remote: Reusing existing pack: 1857, done.
remote: Total 1857 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (1857/1857), 374.35 KiB | 193.00 KiB/s, done.
Resolving deltas: 100% (772/772), done.
Checking connectivity... done.
$ cd ticgit
$ git remote
origin
```

-v옵션을 주어 단축이름과 URL을 함께 볼 수 있다:

```
$ git remote -v
origin  git://github.com/schacon/ticgit.git (fetch)
origin  git://github.com/schacon/ticgit.git (push)
```

리모트 저장소가 여러 개 있다면 이 명령은 전부 보여준다. 내 Grit 저장소에서 실행하면 아래와 같이 출력한다:

```
$ cd grit
$ git remote -v
bakkdoor  git://github.com/bakkdoor/grit.git
cho45     git://github.com/cho45/grit.git
defunkt   git://github.com/defunkt/grit.git
koke      git://github.com/koke/grit.git
origin    git@github.com:mojombo/grit.git
```

이렇게 리모트 저장소가 여러 개가 등록되어 있으면 다른 사람이 기여한 내용(Contributions)을 쉽게 가져올 수 있다. 그리고 origin만 SSH URL이기 때문에 origin에만 Push할 수 있다(4장에서 좀 더 자세히 다룬다).

2.5.2 리모트 저장소 추가하기

이전 절에서도 리모트 저장소를 추가하는 것에 대해 설명했었지만 수박 겉핥기식으로 살펴봤을 뿐이었다. 여기에서는 리모트 저장소를 추가하는 방법을 자세하게 설명한다. 쉽게 새 리모트 저장소를 추가할 수 있는데 `git remote add [단축이름] [url]` 명령을 실행한다:

```
$ git remote
origin
```

```
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin git://github.com/schacon/ticgit.git
pb git://github.com/paulboone/ticgit.git
```

이제 URL 대신에 스트링 pb를 사용할 수 있다. 예를 들어 로컬 저장소에는 없지만 Paul의 저장소에 있는 것을 가져오려면 아래과 같이 실행한다:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

로컬에서 pb/master가 Paul의 master 브랜치이다. 이것을 로컬 브랜치중 하나에 머지하거나 체크아웃하여 브랜치 내용을 자세히 확인할 수 있다.

2.5.3 리모트 저장소를 Pull 하거나 Fetch 하기

앞서 설명했듯이 리모트 저장소에서 데이터를 가져오려면 간단히 아래와 같이 실행한다:

```
$ git fetch [remote-name]
```

이 명령은 로컬에는 없지만, 리모트 저장소에는 있는 데이터를 모두 가져온다. 그리고 나면 리모트 저장소의 모든 브랜치를 로컬에서 접근할 수 있어서 언제든지 머지를 하거나 내용을 살펴볼 수 있다(우리는 3장에서 브랜치를 사용하는 방법에 대해 좀 더 자세히 설명할 것이다).

저장소를 Clone하면 명령은 자동으로 리모트 저장소를 origin이라는 이름으로 추가한다. 그래서 나중에 `git fetch origin`을 실행하면 Clone한 이후에(혹은 마지막으로 가져온 이후에) 수정된 것을 모두 가져온다. `fetch` 명령은 리모트 저장소의 데이터를 모두 로컬로 가져오지만, 자동으로 머지하지 않는다. 그래서 당신이 로컬에서 하던 작업을 정리하고 나서 수동으로 머지해야 한다.

그냥 쉽게 `git pull` 명령으로 리모트 저장소 브랜치에서 데이터를 가져올 뿐만 아니라 자동으로 로컬 브랜치와 머지시킬 수 있다. 먼저 `git clone` 명령은 자동으로 로컬의 master 브랜치가 리모트 저장소의 master 브랜치를 추적하도록 한다(물론 리모트 저장소에 master 브랜치가 있다고 가정에서). 그리고 `git pull` 명령은 Clone한 서버에서 데이터를 가져오고 그 데이터를 자동으로 현재 작업하는 코드와 머지시킨다.

2.5.4 리모트 저장소에 Push하기

프로젝트를 공유하고 싶을 때 리모트 저장소에 Push할 수 있다. 이 명령은 `git push [리모트 저장소 이름] [브랜치 이름]`으로 단순하다. master 브랜치를 `origin` 서버에 Push하려면(다시 말하지만 Clone

하면 보통 자동으로 origin 이름이 생성된다) 아래와 같이 서버에 Push한다:

```
$ git push origin master
```

이 명령은 Clone한 리모트 저장소에 쓰기 권한이 있고, Clone하고 난 이후 아무도 리모트 저장소에 Push하지 않았을 때만 사용할 수 있다. 다시 말해서 Clone한 사람이 여러 명 있을 때, 다른 사람이 Push한 후에 Push하려고 하면 Push할 수 없다. 먼저 다른 사람이 작업한 것을 가져와서 머지한 후에 Push할 수 있다. 3장에서 서버에 Push하는 방법에 대해 자세히 설명할 것이다.

2.5.5 리모트 저장소 살펴보기

(역주, 이 절은 최신 버전의 Git이 출력하는 메시지와 조금 다르다.)
`git remote show [리모트 저장소 이름]` 명령으로 리모트 저장소의 구체적인 정보를 확인할 수 있다. `origin` 같은 단축이름으로 이 명령을 실행하면 아래와 같은 정보를 볼 수 있다:

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

리모트 저장소의 URL과 추적하는 브랜치를 출력한다. 이 명령은 `git pull` 명령을 실행할 때 master 브랜치와 머지할 브랜치가 무엇인지 보여 준다. `git pull` 명령은 리모트 저장소 브랜치의 데이터를 모두 가져오고 나서 자동으로 머지할 것이다. 그리고 가져온 모든 리모트 저장소 정보도 출력한다.
좀 더 Git을 열심히 사용하게 되면 `git remote show` 명령은 더 많은 정보를 보여줄 것이다. 여러분도 언젠가는 아래와 같은 메시지(역주, 다수의 브랜치를 사용하는 메시지)를 볼 날이 올 것이다.

```
$ git remote show origin
* remote origin
  URL: git@github.com:defunkt/github.git
  Remote branch merged with 'git pull' while on branch issues
    issues
  Remote branch merged with 'git pull' while on branch master
    master
  New remote branches (next fetch will store in remotes/origin)
    caching
  Stale tracking branches (use 'git remote prune')
    libwalker
    walker2
```

```
  Tracked remote branches
    acl
    apiv2
    dashboard2
    issues
    master
    postgres
  Local branch pushed with 'git push'
    master:master
```

브랜치명을 생략하고 `git push` 명령을 실행할 때 어떤 브랜치가 어떤 브랜치로 Push되는지 보여준다. 또 아직 로컬로 가져오지 않은 리모트 저장소의 브랜치는 어떤 것들이 있는지, 서버에서는 삭제됐지만 아직 가지고 있는 브랜치는 어떤 것인지, `git pull` 명령을 실행했을 때 자동으로 머지할 브랜치는 어떤 것이 있는지 보여준다.

2.5.6 리모트 저장소 이름을 바꾸거나 리모트 저장소를 삭제하기

`git remote rename` 명령으로 리모트 저장소의 이름을 변경할 수 있다. 예를 들어 `pb`를 `paul`로 변경하려면 `git remote rename` 명령을 사용한다:

```
$ git remote rename pb paul
$ git remote
origin
paul
```

리모트 저장소의 브랜치 이름도 바뀐다. 여태까지 `pb/master`로 리모트 저장소 브랜치를 사용했으면 이제는 `paul/master`라고 사용해야 한다.
리모트 저장소를 삭제해야 한다면 `git remote rm` 명령을 사용한다. 서버 정보가 바뀌었을 때, 더는 별도의 미러가 필요하지 않을 때, 더는 기여자가 활동하지 않을 때 필요하다:

```
$ git remote rm paul
$ git remote
origin
```

2.6 태그

다른 VCS처럼 Git도 태그를 지원한다. 사람들은 보통 릴리즈할 때 사용한다(v1.0, 등등). 이번에는 태그를 조회하고 생성하는 법과 태그의 종류를 설명한다.

2.6.1 태그 조회하기

우선 `git tag` 명령으로 이미 만들어진 태그가 있는지 확인할 수 있다:

```
$ git tag
v0.1
v1.3
```

이 명령은 알파벳 순서로 태그를 보여준다. 사실 순서는 별로 중요한 게 아니다.

검색 패턴을 사용하여 태그를 검색할 수 있다. Git 소스 저장소는 240여 개의 태그가 있다. 만약 1.4.2 버전의 태그들만 검색하고 싶으면 아래와 같이 실행한다:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

2.6.2 태그 붙이기

Git의 태그는 Lightweight 태그와 Annotated 태그로 두 종류가 있다. Lightweight 태그는 브랜치와 비슷한데 브랜치처럼 가리키는 지점을 최신 커밋으로 이동시키지 않는다. 단순히 특정 커밋에 대한 포인터일 뿐이다. 한편, Annotated 태그는 Git 데이터베이스에 태그를 만든 사람의 이름, 이메일과 태그를 만든 날짜, 그리고 태그 메시지도 저장한다. 또 GPG(GNU Privacy Guard)로 서명할 수도 있다. 이 모든 정보를 저장해둬야 할 때에만 Annotated 태그를 추천한다. 그냥 다른 정보를 저장하지 않는 단순한 태그가 필요하다면 Lightweight 태그를 사용하는 것이 좋다.

2.6.3 Annotated 태그

Annotated 태그를 만드는 방법은 간단하다. `tag` 명령을 실행할 때 `-a` 옵션을 추가한다:

```
$ git tag -a v1.4 -m 'my version 1.4'
$ git tag
v0.1
v1.3
v1.4
```

`-m` 옵션으로 태그를 저장할 때 메시지를 함께 저장할 수 있다. 명령을 실행할 때 메시지를 입력하지 않으면 Git은 편집기를 실행시킨다.

`git show` 명령으로 태그 정보와 커밋 정보를 모두 확인할 수 있다:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
```

```
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4

commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

커밋 정보를 보여주기 전에 먼저 태그를 만든 사람이 누구인지, 언제 태그를 만들었는지, 그리고 태그 메시지가 무엇인지 보여준다.

2.6.4 태그에 서명하기

GPG 개인키가 있으면 태그에 서명할 수 있다. 이때에는 `-a`옵션 대신 `-s`를 사용한다:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

이 태그에 `git show`를 실행하면 GPG 서명도 볼 수 있다:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)

iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800
```

```
    Merge branch 'experiment'
```

잠시 후에 서명한 태그를 검증하는 방법도 설명한다.

2.6.5 Lightweight 태그

Lightweight 태그는 기본적으로 파일에 커밋 체크섬을 저장하는 것뿐이다. 다른 정보는 저장하지 않는다. Lightweight 태그를 만들 때에는 `-a`, `-s`, `-m` 옵션을 사용하지 않는다:

```
$ git tag v1.4-lw
$ git tag
v0.1
v1.3
v1.4
v1.4-lw
v1.5
```

이 태그에 `git show`를 실행하면 별도의 태그 정보를 확인할 수 없다. 이 명령은 단순히 커밋 정보만을 보여준다:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

2.6.6 태그 검증하기

`git tag -v [태그 이름]` 명령으로 서명한 태그를 검증한다. 이 명령은 GPG를 사용하여 서명을 검증한다. 그래서 서명자의 GPG 공개키가 필요하다. 이 공개키가 Keyring에 있어야만 이 명령이 성공적으로 실행된다:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
```

```
GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                 aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

만약 서명자의 공개키가 없으면 아래와 같은 메시지를 출력한다:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

2.6.7 나중에 태그하기

예전 커밋에 대해서도 태그할 수 있다. 커밋 히스토리는 아래와 같고:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

"updated rakefile" 커밋을 v1.2로 태그하지 못했다고 해도 차후에 태그를 붙일 수 있다. 특정 커밋에 태그하기 위해서 명령의 끝에 커밋 체크섬을 명시한다(긴 체크섬을 전부 사용할 필요는 없다):

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

이제 아래와 같이 만든 태그를 확인한다:

```
$ git tag
v0.1
v1.2
v1.3
v1.4
v1.4-lw
v1.5

$ git show v1.2
tag v1.2
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:32:16 2009 -0800

version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

    updated rakefile
...
```

2.6.8 태그 공유하기

`git push` 명령은 자동으로 리모트 서버에 태그를 전송하지 않는다. 태그를 만들었으면 서버에 별도로 Push해야 한다. 브랜치를 공유하는 것과 같은 방법으로 할 수 있다. `git push origin [태그 이름]`을 실행한다:

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag]         v1.5 -> v1.5
```

만약 한 번에 태그를 여러 개 Push하고 싶으면 `--tags` 옵션을 추가하여 `git push`명령을 실행한다. 이 명령으로 리모트 서버에 없는 태그를 모두 전송할 수 있다:

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
```

```
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
 * [new tag]         v0.1 -> v0.1
 * [new tag]         v1.2 -> v1.2
 * [new tag]         v1.4 -> v1.4
 * [new tag]         v1.4-lw -> v1.4-lw
 * [new tag]         v1.5 -> v1.5
```

누군가 저장소에서 Clone하거나 Pull을 하면 모든 태그 정보도 함께 전송된다.

2.7 팁과 트릭

Git의 기초를 마치기 전에 Git을 좀 더 쉽고 편안하게 쓸 수 있게 만들어 줄 몇 가지 팁과 트릭도 설명한다. 이런 팁 없이 Git을 사용하는 사람들도 많다. 우리는 이 책에서 이 팁을 다시 거론하지 않고 이런 팁을 알고 있다고 가정한다. 그래서 알고 있는 것이 좋다.

2.7.1 자동완성

Bash 쉘을 쓰고 있다면 멋진 자동완성(Auto-completion) 기능을 사용할 수 있다. https://github.com/git/git/blob/master/contrib/completion/git-completion.bash 에서 바로 다운받는다. 그 파일을 홈 디렉토리에 카피하고 `.bashrc` 파일에 아래와 같은 내용을 추가하자:

```
source ~/git-completion.bash
```

또 모든 사용자가 사용할 수 있게 설정할 수 있다. Mac 시스템이라면 이 스크립트를 `/opt/local/etc/bash_completion.d` 디렉토리에 복사하고 리눅스라면 `/etc/bash_completion.d/`에 복사한다. 이 디렉토리는 Bash가 자동완성을 지원하기 위해 사용하는 디렉토리다.
윈도에 msysGit을 설치해서 Git Bash를 사용하는 경우에는 자동완성이 미리 설정되어 있다.
Git 명령을 입력할 때 <Tab> 키를 누르면 Git이 제안하는 명령어가 출력된다:

```
$ git co<tab><tab>
commit config
```

이 경우 `git co`를 입력하고 Tab 키를 두번 누르면 commit과 config를 제안한다. 이 때 `m<tab>`을 입력하면 자동으로 `git commit`명령을 완성한다.
옵션에도 이 기능이 되고 더 유용하다. 예를 들어 `git log`명령을 실행하는데 옵션이 전혀 기억나지 않는다면 아래와 같이 입력하고 Tap 키를 누르면 아래와 같은 옵션을 제안한다:

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

이건 상당히 멋진 팁이다. 아마 문서를 찾아보는 등의 시간을 절약해 줄 것이다.

2.7.2 Git Alias

명령을 완벽하게 입력하지 않으면 Git은 알아듣지 못한다. Git의 명령을 전부 입력하는 것이 귀찮다면 `git config`를 사용하여 각 명령의 Alias을 쉽게 만들 수 있다. 아래는 Alias을 만드는 예이다:

```
$ git config --global alias.co checkout
$ git config --global alias.br branch
$ git config --global alias.ci commit
$ git config --global alias.st status
```

이제 `git commit` 대신 `git ci`만으로도 커밋할 수 있다. Git을 계속 사용한다면 다른 명령어도 자주 사용하게 될 것이다. 자주 사용하는 명령은 Alias을 만들어 편하게 사용한다.
이미 있는 명령을 편리하고 새로운 명령으로 만들어 사용할 수 있다. 예를 들어 파일을 Unstage 상태로 변경하는 명령을 만들어서 불편함을 덜 수 있다. 아래와 같이 `unstage` 라는 Alias을 만든다:

```
$ git config --global alias.unstage 'reset HEAD --'
```

아래 두 명령은 동일한 명령이다:

```
$ git unstage fileA
$ git reset HEAD fileA
```

한결 간결해졌다. 추가로 `last` 명령을 만들어 보자:

```
$ git config --global alias.last 'log -1 HEAD'
```

이제 최근 커밋을 좀 더 쉽게 확인할 수 있다:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

이것으로 쉽게 새로운 명령을 만들 수 있다. 그리고 Git의 명령어뿐만 아니라 외부 명령어도 실행할 수 있다. `!`를 제일 앞에 추가하면 외부 명령을 실행한다. 아래 명령은 `git visual`이라고 입력하면 `gitk`가 실행된다:

```
$ git config --global alias.visual '!gitk'
```

2.8 요약

이제 우리는 로컬에서 사용할 수 있는 Git 명령에 대한 기본 지식은 갖추었다. 저장소를 만들고 Clone하는 방법, 수정하고 나서 Stage하고 커밋하는 방법, 저장소의 히스토리를 조회하는 방법 등을 살펴보았다. 이어지는 장에서는 Git의 가장 강력한 기능인 브랜치 모델을 살펴볼 것이다.

3장

Git 브랜치

모든 버전 관리 시스템은 브랜치를 지원한다. 개발을 하다 보면 코드를 여러 개로 복사해야 하는 일이 자주 생긴다. 코드를 통째로 복사하고 나서 원래 코드와는 상관없이 독립적으로 개발을 진행할 수 있는데, 이렇게 독립적으로 개발하는 것이 브랜치다.
버전 관리 시스템에서 브랜치를 만드는 과정은 고생스럽다. 개발자가 수동으로 소스코드 디렉토리를 복사해서 브랜치를 만들어야 하고 소스코드의 양이 많으면 브랜치를 만드는 시간도 오래 걸린다.
사람들은 브랜치 모델이 Git의 최고의 장점이라고, Git이 다른 것들과 구분되는 특징이라고 말한다. 당최 어떤 점이 그렇게 특별한 것일까? Git의 브랜치는 매우 가볍다. 순식간에 브랜치를 새로 만들고 브랜치 사이를 이동할 수 있다. 다른 버전 관리 시스템과는 달리 Git은 브랜치를 만들어 작업하고 나중에 Merge하는 방법을 권장한다. 심지어 하루에 수십 번씩해도 괜찮다. Git 브랜치에 능숙해지면 개발 방식이 완전히 바뀌고 다른 도구를 사용할 수 없게 된다.

3.1 브랜치란 무엇인가?

Git이 브랜치하는 과정을 이해하려면 우선 Git이 데이터를 어떻게 저장하는지 알아야 한다. Git은 데이터를 `Change Set`이나 변경사항(Diff)으로 기록하지 않고 일련의 스냅샷으로 기록한다는 것을 1장에서 보여줬다.
커밋하면 Git은 현 Staging Area에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타데이터, 이전 커밋에 대한 포인터 등을 포함하는 커밋 개체(커밋 Object)를 저장한다. 이전 커밋 포인터가 있어서 현재 커밋이 무엇을 기준으로 바뀌었는지를 알 수 있다. 최초 커밋을 제외한 나머지 커밋은 이전 커밋 포인터가 적어도 하나씩 있고 브랜치를 합친 Merge 커밋 같은 경우에는 이전 커밋 포인터가 여러 개 있다.
예제를 보자. 파일이 3개 있는 디렉토리가 하나 있고 이 파일을 Staging Area에 저장하고 커밋해 보자. 파일을 Stage하면 Git 저장소에 파일을 저장하고(Git은 이것을 Blob이라고 부른다) Staging Area에 해당 파일의 체크섬을 저장한다(1장에서 살펴본 SHA-1을 사용한다).

```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

'git commit'으로 커밋하면 먼저 루트 디렉토리와 각 하위 디렉토리의 트리 개체를 체크섬과 함께 저장소에 저장한다. 그다음에 커밋 개체를 만들고 메타데이터와 루트 디렉토리 트리 개체를 가리키는 포인터 정보를 커밋 개체에 넣어 저장한다. 그래서 필요하면 언제든지 스냅샷을 다시 만들 수 있다.

이 작업을 마치고 나면 Git 저장소에는 다섯 개의 데이터 개체가 생긴다. 각 파일에 대한 Blob 세 개, 파일과 디렉토리 구조가 들어 있는 트리 개체 하나, 메타데이터와 루트 트리를 가리키는 포인터가 담긴 커밋 개체 하나이다. 이것을 그림으로 그리면 그림 3-1과 같다.

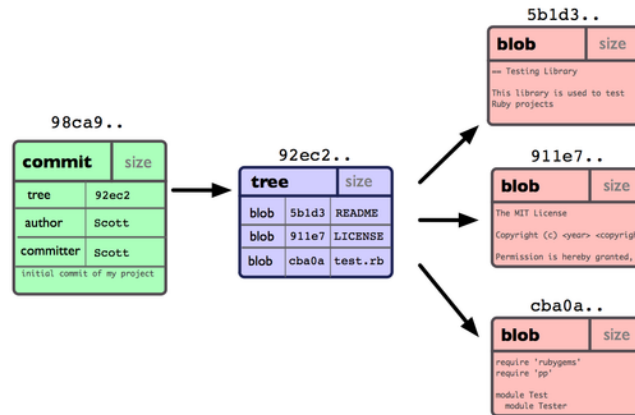



그림 3.1: 저장소의 커밋 데이터

다시 파일을 수정하고 커밋하면 이전 커밋이 무엇인지도 저장한다. 커밋을 두 번 더 하면 그림 3-2과 같이 저장된다.

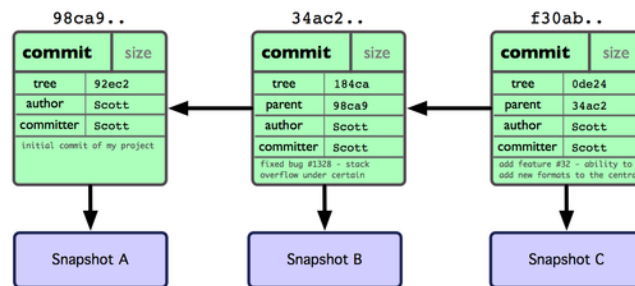



그림 3.2: Git 커밋의 개체 데이터

Git의 브랜치는 커밋 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것이다. 기본적으로 Git은 `master` 브랜치를 만든다. 최초로 커밋하면 Git은 master라는 이름의 브랜치를 만들어서 자동으로 가장 마지막 커밋을 가리키게 한다.

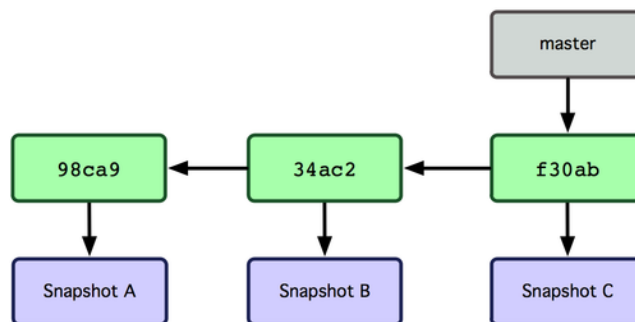



그림 3.3: 가장 최근 커밋 정보를 가리키는 브랜치

브랜치를 하나 새로 만들면 어떨까? 브랜치를 하나 만들어서 놀자. 다음과 같이 `git branch` 명령으로 testing 브랜치를 만든다.

```
$ git branch testing
```

새로 만든 브랜치도 지금 작업하고 있던 마지막 커밋을 가리킨다(그림 3-4).

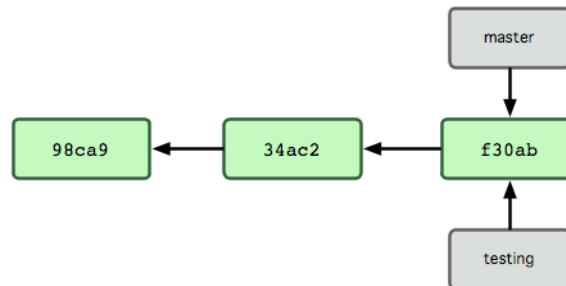



그림 3.4: 커밋 개체를 가리키는 두 브랜치

지금 작업 중인 브랜치가 무엇인지 Git은 어떻게 파악할까? 다른 버전 관리 시스템과는 달리 Git은 'HEAD'라는 특수한 포인터가 있다. 이 포인터는 지금 작업하는 로컬 브랜치를 가리킨다. 브랜치를 새로 만들었지만, Git은 아직 master 브랜치를 가리키고 있다. `git branch` 명령은 브랜치를 만들기만 하고 브랜치를 옮기지 않는다.

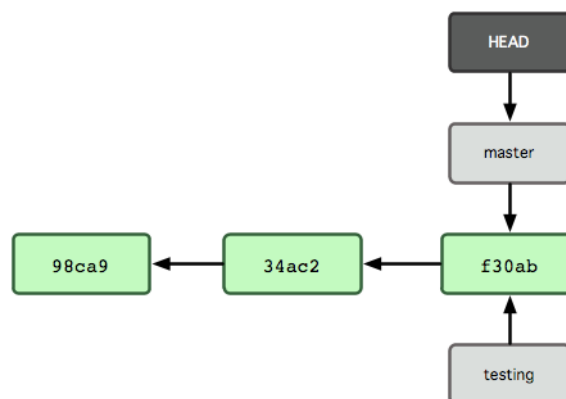



그림 3.5: HEAD는 현재 작업 중인 브랜치를 가리킴

`git checkout` 명령으로 새로 만든 브랜치로 이동할 수 있다. testing 브랜치로 이동하려면 다음과 같이 한다:

```
$ git checkout testing
```

이렇게 하면 HEAD는 testing 브랜치를 가리킨다.
자, 이제 핵심이 보일 거다! 커밋을 새로 한 번 해보면:

```
$ vim test.rb
$ git commit -a -m 'made a change'
```

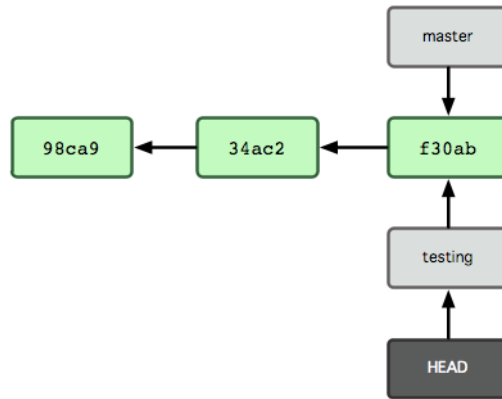



그림 3.6: HEAD는 옮겨간 다른 브랜치를 가리킨다

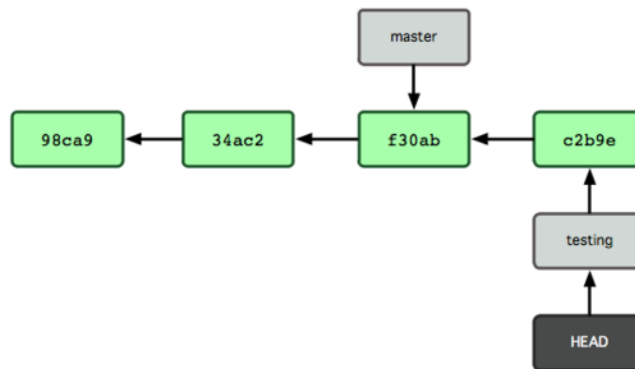



그림 3.7: HEAD가 가리키는 testing 브랜치가 새 커밋을 가리킨다

결과는 그림 3-7과 같다.
이 부분이 흥미롭다. 새로 커밋해서 testing 브랜치는 앞으로 이동했다. 하지만, `master` 브랜치는 여전히 이전 커밋을 가리킨다. `master` 브랜치로 되돌아가면:

```
$ git checkout master
```

결과는 그림 3-8과 같다.

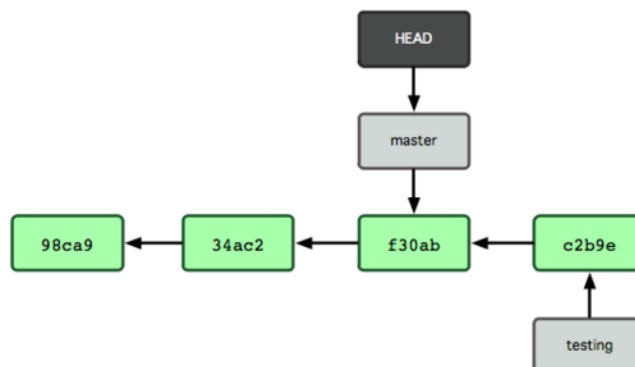



그림 3.8: HEAD가 Checkout한 브랜치로 이동함

방금 실행한 명령이 한 일은 두 가지다. master 브랜치가 가리키는 커밋을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점으로 되돌려 놓았다. 앞으로 커밋을 하면 다른 브랜치의 작업들과 별개로

진행되기 때문에 testing 브랜치에서 임시로 작업하고 원래 master 브랜치로 돌아와서 하던 일을 계속 할 수 있다.

파일을 수정하고 다시 커밋을 해보자:

```
$ vim test.rb
$ git commit -a -m 'made other changes'
```

프로젝트 히스토리는 분리돼 진행한다(그림 3-9). 우리는 브랜치를 하나 만들어 그 브랜치에서 일을 좀 하고, 다시 원래 브랜치로 되돌아와서 다른 일을 했다. 두 작업 내용은 서로 독립적으로 각 브랜치에 존재한다. 커밋 사이를 자유롭게 이동하다가 때가 되면 두 브랜치를 Merge한다. 간단히 `branch`와 `checkout` 명령을 써서 말이다.

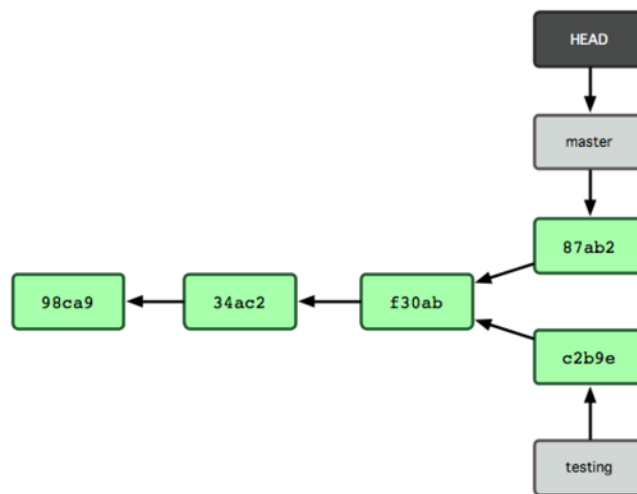



그림 3.9: 브랜치 히스토리가 서로 독립적임

실제로 Git의 브랜치는 어떤 한 커밋을 가리키는 40글자의 SHA-1 체크섬 파일에 불과하기 때문에 만들기도 쉽고 지우기도 쉽다. 새로 브랜치를 하나 만드는 것은 41바이트 크기의 파일을(40자와 줄 바꿈 문자) 하나 만드는 것에 불과하다.

브랜치를 만들어야 하면 프로젝트를 통째로 복사해야 하는 다른 버전 관리 도구와 Git의 차이는 극명하다. 통째로 복사하는 작업은 프로젝트 크기에 따라 다르겠지만 수십 초에서 수십 분까지 걸린다. 그에 비해 Git은 순식간이다. 게다가 커밋을 할 때마다 이전 커밋의 정보를 저장하기 때문에 Merge할 때 어디서부터(Merge Base) 합쳐야 하는지 안다. 이런 특징은 개발자들이 수시로 브랜치를 만들어 사용하게 한다.

이제 왜 그렇게 브랜치를 수시로 만들고 사용해야 하는지 알아보자.

3.2 브랜치와 Merge의 기초

실제 개발과정에서 겪을 만한 예제를 하나 살펴보자. 브랜치와 Merge는 보통 이런 식으로 진행한다:

1. 작업 중인 웹사이트가 있다.
2. 새로운 이슈를 처리할 새 Branch를 하나 생성.
3. 새로 만든 Branch에서 작업 중.

이때 중요한 문제가 생겨서 그것을 해결하는 Hotfix를 먼저 만들어야 한다. 그러면 다음과 같이 할 수 있다:

1. 새로운 이슈를 처리하기 이전의 운영(Production) 브랜치로 이동.
2. Hotfix 브랜치를 새로 하나 생성.
3. 수정한 Hotfix 테스트를 마치고 운영 브랜치로 Merge.
4. 다시 작업하던 브랜치로 옮겨가서 하던 일 진행.

3.2.1 브랜치의 기초

먼저 커밋을 몇 번 했다고 가정하자.

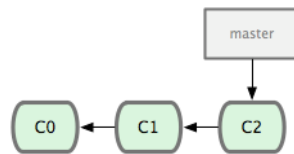



그림 3.10: 현재 커밋 히스토리

이슈 관리 시스템에 등록된 53번 이슈를 처리한다고 하면 이 이슈에 집중할 수 있는 브랜치를 새로 하나 만든다. Git은 어떤 이슈 관리 시스템에도 종속돼 있지 않다. 브랜치를 만들면서 Checkout까지 한 번에 하려면 `git checkout` 명령에 -b라는 옵션을 준다.

```
$ git checkout -b iss53
Switched to a new branch 'iss53'
```

위 명령은 아래 명령을 줄여놓은 것이다:

```
$ git branch iss53
$ git checkout iss53
```

그림 3-11은 위 명령의 결과를 나타낸다.

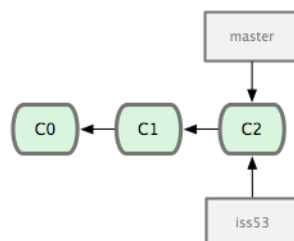



그림 3.11: 브랜치 포인터를 새로 만듦

iss53 브랜치를 Checkout했기 때문에(즉, HEAD는 iss53 브랜치를 가리킨다) 뭔가 일을 하고 커밋하면 iss53 브랜치가 앞으로 진행한다:

```
$ vim index.html
$ git commit -a -m 'added a new footer [issue 53]'
```

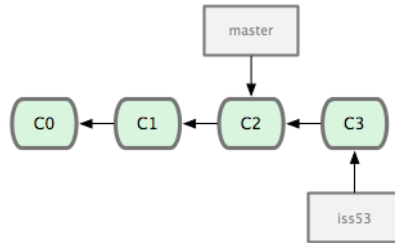



그림 3.12: 진행 중인 iss53 브랜치

다른 상황을 가정해보자. 만드는 사이트에 문제가 생겨서 즉시 고쳐야 한다. 버그를 해결한 Hotfix에 'iss53'이 섞이는 것을 방지하기 위해 'iss53'와 관련된 코드를 어딘가에 저장해두고 원래 운영 환경의 소스로 복구해야 한다. Git을 사용하면 이런 노력을 들일 필요 없이 그냥 master 브랜치로 옮기면 된다.

그렇지만, 브랜치를 이동하려면 해야 할 일이 있다. 아직 커밋하지 않은 파일이 Checkout할 브랜치와 충돌 나면 브랜치를 변경할 수 없다. 브랜치를 변경할 때에는 워킹 디렉토리를 정리하는 것이 좋다. 이런 문제를 다루는 방법은(주로, Stash이나 커밋 Amend에 대해) 나중에 다룰 것이다. 지금은 작업하던 것을 모두 커밋하고 master 브랜치로 옮긴다:

```
$ git checkout master
Switched to branch 'master'
```

이때 워킹 디렉토리는 53번 이슈를 시작하기 이전 모습으로 되돌려지기 때문에 새로운 문제에 집중할 수 있는 환경이 만들어진다. Git은 자동으로 워킹 디렉토리에 파일들을 추가하고, 지우고, 수정해서 Checkout한 브랜치의 스냅샷으로 되돌려 놓는다는 것을 기억해야 한다.

hotfix라는 브랜치를 만들고 새로운 이슈를 해결할 때까지 사용한다:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix 3a0874c] fixed the broken email address
 1 files changed, 1 deletion(-)
```

운영 환경에 적용하려면 문제를 제대로 고쳤는지 테스트하고 master 브랜치에 합쳐야 한다. `git merge` 명령으로 다음과 같이 한다:

```
$ git checkout master
```

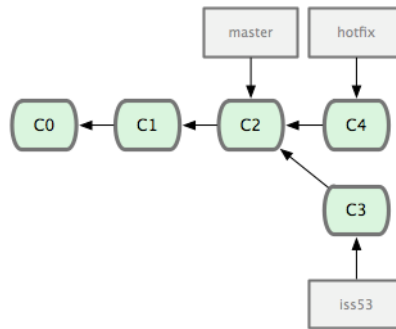



그림 3.13: master 브랜치에서 갈라져 나온 hotfix 브랜치

```
$ git merge hotfix
Updating f42c576..3a0874c
Fast-forward
 README | 1 -
 1 file changed, 1 deletion(-)
```

Merge 메시지에서 'Fast-forward'가 보이는가? Merge할 브랜치가 가리키고 있던 커밋이 현 브랜치가 가리키는 것보다 '앞으로 진행한' 커밋이기 때문에 master 브랜치 포인터는 최신 커밋으로 이동한다. 이런 Merge 방식을 'Fast forward'라고 부른다. 다시 말해서 A 브랜치에서 다른 B 브랜치를 Merge할 때 B가 A 이후의 커밋을 가리키고 있으면 그저 A가 B의 커밋을 가리키게 할 뿐이다.

이제 hotfix는 master 브랜치에 포함됐고 운영환경에 적용할 수 있다(그림 3-14).

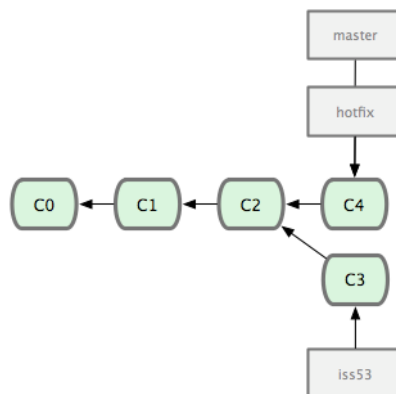



그림 3.14: Merge 후 hotfix 브랜치와 같은 것을 가리키는 master 브랜치

문제를 급히 해결하고 master 브랜치에 적용하고 나면 다시 일하던 브랜치로 돌아가야 한다. 하지만, 그전에 필요없는 hotfix 브랜치를 삭제한다. `git branch` 명령에 `-d` 옵션을 주고 브랜치를 삭제한다.

```
$ git branch -d hotfix
Deleted branch hotfix (was 3a0874c).
```

자 이제 이슈 53번을 처리하던 환경으로 되돌아가서 하던 일을 계속 하자(그림 3-15):

```
$ git checkout iss53
Switched to branch 'iss53'
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53 ad82d7a] finished the new footer [issue 53]
 1 file changed, 1 insertion(+)
```

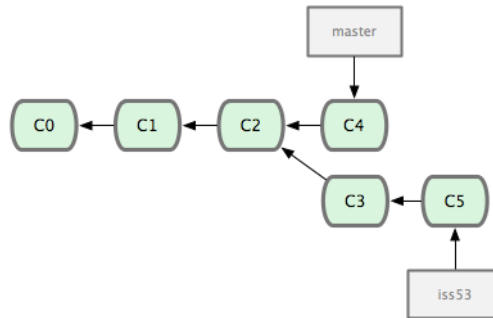



그림 3.15: master와 별개로 진행하는 iss53 브랜치

위에서 작업한 hotfix가 iss53 브랜치에 영향을 끼치지 않는다는 점을 이해하는 것이 중요하다. `git merge master` 명령으로 master 브랜치를 iss53 브랜치에 Merge하면 iss53 브랜치에 hotfix가 적용된다. 아니면 iss53 브랜치가 master에 Merge할 수 있는 수준이 될 때까지 기다렸다가 Merge하면 hotfix와 iss53가 합쳐진다.

3.2.2 Merge의 기초

53번 이슈를 다 구현하고 master 브랜치에 Merge하는 과정을 살펴보자. master 브랜치에 Merge하는 것은 앞서 살펴본 hotfix 브랜치를 Merge하는 것과 비슷하다. `git merge` 명령으로 합칠 브랜치에서 합쳐질 브랜치를 Merge하면 된다:

```
$ git checkout master
$ git merge iss53
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 1 +
 1 file changed, 1 insertion(+)
```

hotfix를 Merge했을 때와 메시지가 다르다. 현 브랜치가 가리키는 커밋이 Merge할 브랜치의 조상이 아니므로 Git은 'Fast-forward'로 Merge하지 않는다. 이러면 Git은 각 브랜치가 가리키는 커밋 두 개와 공통 조상 하나를 사용하여 3-way Merge를 한다. 그림 3-16에 이 Merge에서 사용하는 커밋 세 개가 표시된다.

단순히 브랜치 포인터를 최신 커밋으로 옮기는 게 아니라 3-way Merge의 결과를 별도의 커밋으로 만들고 나서 해당 브랜치가 그 커밋을 가리키도록 이동시킨다(그림 3-17). 그래서 이런 커밋은 부모가 여러 개고 Merge 커밋이라고 부른다.

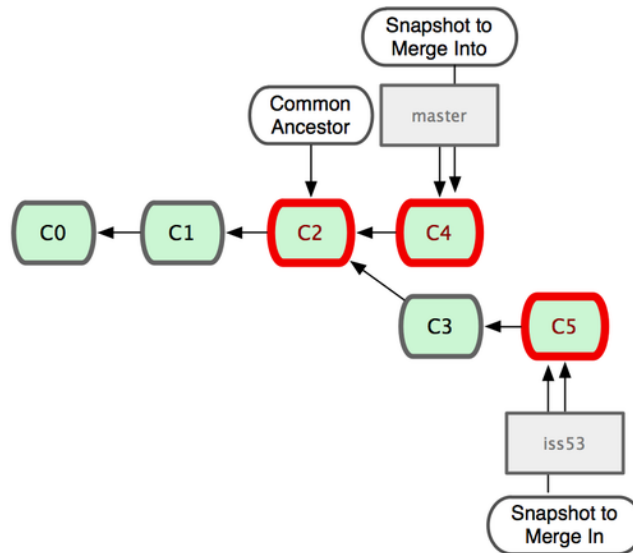



그림 3.16: Git은 Merge에 필요한 공통 커밋을 자동으로 찾음

Git은 Merge하는데 필요한 최적의 공통 조상을 자동으로 찾는다. 이런 기능도 Git이 다른 버전 관리 시스템보다 나은 점이다. CVS나 Subversion 같은 버전 관리 시스템은 개발자가 직접 공통 조상을 찾아서 Merge해야 한다. Git은 다른 시스템보다 Merge가 대단히 쉽다.

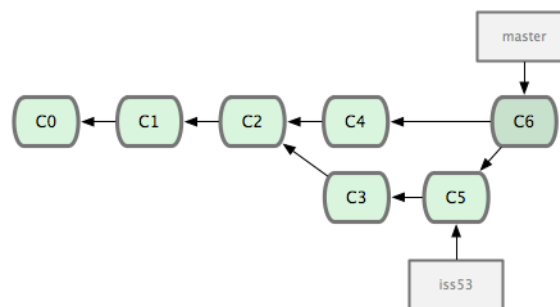



그림 3.17: Git은 Merge할 때 Merge에 대한 정보가 들어 있는 커밋를 하나 만든다.

iss53 브랜치를 master에 Merge하고 나면 더는 iss53 브랜치가 필요 없다. 다음 명령으로 브랜치를 삭제하고 이슈의 상태를 처리 완료로 표시한다:

```
$ git branch -d iss53
```

3.2.3 충돌의 기초

가끔씩 3-way Merge가 실패할 때도 있다. Merge하는 두 브랜치에서 같은 파일의 한 부분을 동시에 수정하고 Merge하면 Git은 해당 부분을 Merge하지 못한다. 예를 들어, 53번 이슈와 hotfix가 같은 부분을 수정했다면 Git은 Merge하지 못하고 다음과 같은 충돌(Conflict) 메시지를 출력한다:

```
$ git merge iss53
Auto-merging index.html
```

```
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Git은 자동으로 Merge하지 못해서 새 커밋이 생기지 않는다. 변경사항의 충돌을 개발자가 해결하지 않는 한 Merge 과정을 진행할 수 없다. Merge 충돌이 일어났을 때 Git이 어떤 파일을 Merge할 수 없었는지 살펴보려면 `git status` 명령을 이용한다:

```
$ git status
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

충돌이 일어난 파일은 unmerged 상태로 표시된다. Git은 충돌이 난 부분을 표준 형식에 따라 표시해 준다. 그러면 개발자는 해당 부분을 수동으로 해결한다. 충돌 난 부분은 다음과 같이 표시된다.

```
<<<<<<< HEAD
<div id='footer'>contact : email.support@github.com</div>
=======
<div id='footer'>
  please contact us at support@github.com
</div>
>>>>>>> iss53
```

======= 위쪽의 내용은 HEAD 버전(merge 명령을 실행할 때 작업하던 master 브랜치)의 내용이고 아래쪽은 iss53 브랜치의 내용이다. 충돌을 해결하려면 위쪽이나 아래쪽 내용 중에서 고르거나 새로 작성하여 Merge한다. 다음은 아예 새로 작성하여 충돌을 해결하는 예제다:

```
<div id='footer'>
please contact us at email.support@github.com
</div>
```

충돌한 양쪽에서 조금씩 가져와서 새로 수정했다. 그리고 <<<<<<<, =======, >>>>>>> 가 포함된 행을 삭제하였다. 이렇게 충돌한 부분을 해결하고 `git add` 명령으로 다시 Git에 저장한다. 충돌을 쉽게 해결하기 위해 다른 Merge 도구도 이용할 수 있다. `git mergetool` 명령으로 실행한다:

```
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis bc3 codecompare vimdiff
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (opendiff):
```

Mac에서는 `opendiff`가 실행된다. 기본 도구 말고 사용할 수 있는 다른 Merge 도구도 있는데, "… one of the following tools:" 부분에 보여준다. 여기에 표시된 도구 중 하나를 고를 수 있다. Merge 도구를 변경하는 방법은 7장에서 다룬다.

Merge 도구를 종료하면 Git은 잘 Merge했는지 물어본다. 잘 마쳤다고 입력하면 자동으로 `git add`가 수행되고 해당 파일이 Staging Area에 저장된다.

`git status` 명령으로 충돌이 해결된 상태인지 다시 한번 확인해볼 수 있다.

```
$ git status
On branch master
Changes to be committed:
  (use 'git reset HEAD <file>...' to unstage)

        modified:   index.html
```

충돌을 해결하고 나서 해당 파일이 Staging Area에 저장됐는지 확인했으면 `git commit` 명령으로 Merge 한 것을 커밋한다. 충돌을 해결하고 Merge할 때에는 커밋 메시지가 아래와 같다.

```
Merge branch 'iss53'

Conflicts:
  index.html
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
#       .git/MERGE_HEAD
# and try again.
#
```

어떻게 충돌을 해결했고 좀 더 확인해야 하는 부분은 무엇을 어떻게 했는지 자세하게 기록한다. 자세한 기록은 나중에 이 Merge 커밋을 이해하는데 도움을 줄 것이다.

3.3 브랜치 관리

지금까지 브랜치를 만들고, Merge하고, 삭제하는 방법에 대해서 살펴봤다. 브랜치를 관리하는 데 필요한 다른 명령도 살펴보자.
`git branch` 명령은 단순히 브랜치를 만들고 삭제해 주기만 하는 것이 아니다. 아무런 옵션 없이 실행하면 브랜치의 목록을 보여준다:

```
$ git branch
  iss53
* master
  testing
```

* 기호가 붙어 있는 master브랜치는 현재 Checkout해서 작업하는 브랜치를 나타낸다. 즉, 지금 수정한 내용을 커밋하면 master 브랜치에 커밋되고 포인터가 앞으로 한 단계 나아간다. `git branch -v` 명령을 실행하면 브랜치마다 마지막 커밋 메시지도 함께 보여준다:

```
$ git branch -v
  iss53   93b412c fix javascript issue
* master  7a98805 Merge branch 'iss53'
  testing 782fd34 add scott to the author list in the readmes
```

각 브랜치가 지금 어떤 상태인지 확인하기에 좋은 옵션도 있다. 현재 Checkout한 브랜치를 기준으로 Merge된 브랜치인지 그렇지 않은지 필터링해 볼 수 있다. `--merged`와 `--no-merged` 옵션을 사용하여 해당 목록을 볼 수 있다. `git branch --merged` 명령으로 이미 Merge한 브랜치 목록을 확인한다:

```
$ git branch --merged
  iss53
* master
```

iss53 브랜치는 앞에서 이미 Merge했기 때문에 목록에 나타난다. * 기호가 붙어 있지 않은 브랜치는 `git branch -d` 명령으로 삭제해도 되는 브랜치다. 이미 다른 브랜치와 Merge 했기 때문에 삭제해도 정보를 잃지 않는다.
반대로 현재 Checkout한 브랜치에 Merge하지 않은 브랜치를 살펴보려면 `git branch --no-merged` 명령을 사용한다:

```
$ git branch --no-merged
  testing
```

위에는 없었던 다른 브랜치가 보인다. 아직 Merge하지 않은 커밋을 담고 있기 때문에 `git branch -d` 명령으로 삭제되지 않는다:

```
$ git branch -d testing
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Merge하지 않은 브랜치를 강제로 삭제하려면 `-D` 옵션으로 삭제한다.

3.4 브랜치 Workflow

브랜치를 만들고 Merge하는 것을 어디에 써먹어야 할까? 이 절에서는 Git의 브랜치가 유용한 몇 가지 Workflow를 살펴본다. 여기서 설명하는 Workflow를 개발에 적용하면 도움이 될 것이다.

3.4.1 Long-Running 브랜치

Git은 꼼꼼하게 3-way Merge를 사용하기 때문에 장기간에 걸쳐서 한 브랜치를 다른 브랜치와 여러 번 Merge하는 것도 어렵지 않다. 그래서 개발 과정에서 필요한 용도에 따라 브랜치를 만들어 두고 계속 사용할 수 있다. 그리고 정기적으로 브랜치를 다른 브랜치로 Merge한다:

이런 접근법에 따라서 Git 개발자가 많이 선호하는 Workflow가 하나 있다. 배포했거나 배포할 코드만 master 브랜치에 Merge해서 안정 버전의 코드만 master 브랜치에 둔다. 개발을 진행하고 안정화하는 브랜치는 develop이나 next라는 이름으로 추가로 만들어 사용한다. 이 브랜치는 언젠가 안정 상태가 되겠지만, 항상 안정 상태를 유지해야 하는 것이 아니다. 테스트를 거쳐서 안정적이라고 판단되면 master 브랜치에 Merge한다. 토픽 브랜치(앞서 살펴본 iss53 브랜치 같은 짧은 호흡 브랜치)에도 적용할 수 있는데, 해당 토픽을 처리하고 테스트해서 버그도 없고 안정적이면 그때 Merge한다.

사실 우리가 얘기하는 것은 커밋을 가리키는 포인터에 대한 얘기다. 개발 브랜치는 공격적으로 히스토리를 만들어 나아가고 안정 브랜치는 이미 만든 히스토리를 뒤따르며 나아간다.

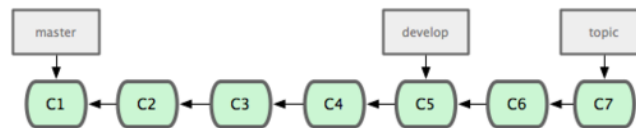



그림 3.18: 안정적인 브랜치일수록 커밋 히스토리가 뒤쳐진다

실험실에서 충분히 테스트하고 실전에 배치하는 과정으로 보면 이해하기 쉽다(그림 3-19).

코드를 여러 단계로 나누어 안정성을 높여가며 운영할 수 있다. 큰 규모의 프로젝트라면 proposed 혹은 pu(proposed updates)라는 이름의 브랜치를 두어 next나 master 브랜치에 아직 Merge할 준비가 되지 않은 것을 일단 Merge시킨다.

중요한 개념은 브랜치를 이용해 여러 단계에 걸쳐서 안정화해 나아가면서 충분히 안정화가 됐을 때 안정 브랜치로 Merge한다는 점이다. 다시 말해서 반드시 Long-Running의 브랜치를 여러 개 만들어야 하는 것은 아니지만 정말 유용하다. 특히 규모가 크고 복잡한 프로젝트일수록 그 유용성이 반짝반짝 빛난다.

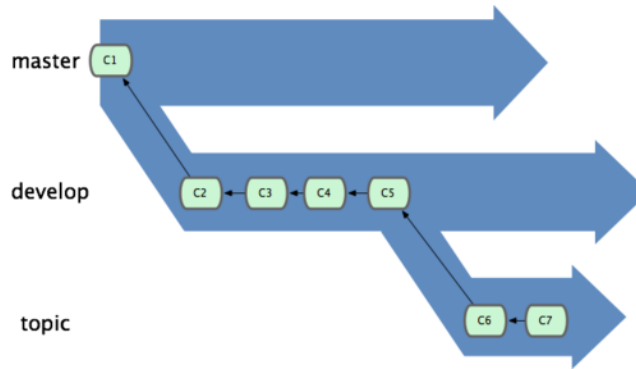



그림 3.19: 각 브랜치를 하나의 실험실로 생각하라

3.4.2 토픽 브랜치

토픽 브랜치는 프로젝트 크기에 상관없이 유용하다. 토픽 브랜치는 어떤 한 가지 주제나 작업을 위해 만든 짧은 호흡의 브랜치다. 다른 버전 관리 시스템에서 이런 브랜치를 본 적이 없을 것이다. Git이 아닌 다른 버전 관리 도구에서는 브랜치를 하나 만드는 데 큰 비용이 든다. Git에서는 매우 일상적으로 브랜치를 만들고 Merge하고 삭제한다.

앞서 사용한 iss53이나 hotfix 브랜치가 토픽 브랜치다. 우리는 브랜치를 새로 만들고 어느 정도 커밋하고 나서 다시 master 브랜치에 Merge하고 브랜치 삭제도 해 보았다. 보통 주제별로 브랜치를 만들고 각각은 독립돼 있기 때문에 매우 쉽게 컨텍스트 사이를 옮겨 다닐 수 있다. 묶음별로 나눠서 일하면 내용별로 검토하기에도, 테스트하기에도 더 편하다. 각 작업을 하루든 한 달이든 유지하다가 master 브랜치에 Merge할 시점이 되면 순서에 관계없이 그때 Merge하면 된다.

master 브랜치를 checkout한 상태에서 어떤 작업을 한다고 해보자. 한 이슈를 처리하기 위해서 iss91라는 브랜치를 만들고 해당 작업을 한다. 같은 이슈를 다른 방법으로 해결해보고 싶을 때도 있다. iss91v2라는 브랜치를 만들고 다른 방법을 시도해 본다. 확신할 수 없는 아이디어를 적용해보기 위해 다시 master 브랜치로 되돌아가서 dumbidea 브랜치를 하나 더 만든다. 지금까지 말했던 커밋 히스토리는 그림 3-20과 같다.

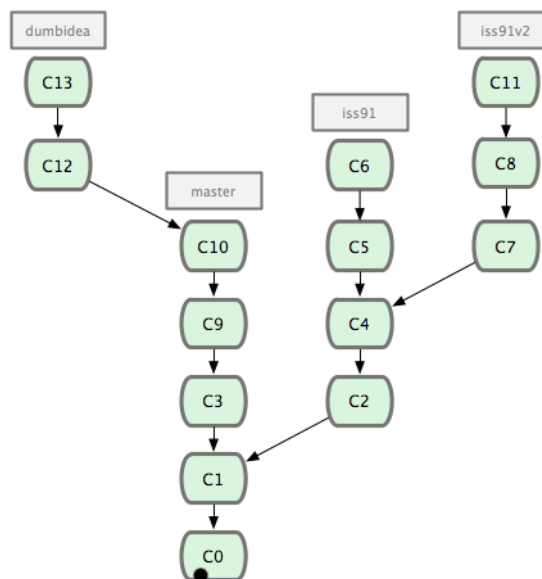



그림 3.20: 여러 토픽 브랜치에 대한 커밋 히스토리

이슈를 처리했던 방법 중 두 번째 방법인 iss91v2 브랜치가 괜찮아서 적용하기로 결정을 내렸다. 그리고 아이디어를 확신할 수 없었던 dumbidea 브랜치를 같이 일하는 다른 개발자에게 보여줬더니 썩 괜

찾다는 반응을 얻었다. iss91 브랜치는 (C5, C6 커밋도 함께) 버리고 다른 두 브랜치를 Merge하면 그림 3-21과 같이 된다.

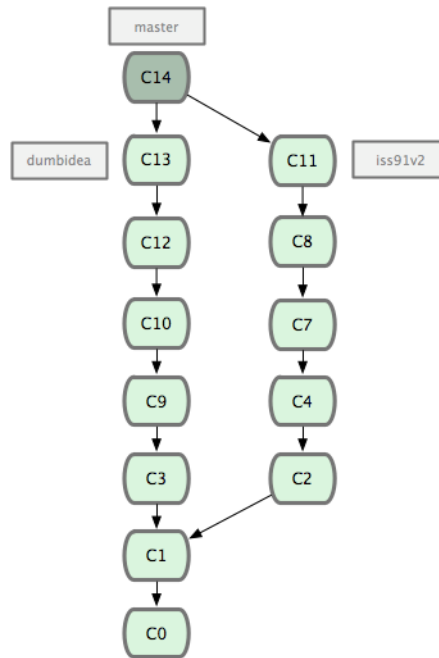



그림 3.21: dumbidea와 iss91v2 브랜치를 Merge하고 난 후의 모습

지금까지 한 작업은 전부 로컬에서만 처리한다는 것을 꼭 기억하자. 로컬 저장소에서만 브랜치를 만들고 Merge했으며 서버와 통신을 주고받는 일은 없었다.

3.5 리모트 브랜치

리모트 브랜치란 리모트 저장소에 있는 브랜치를 말한다. 사실 리모트 브랜치도 로컬에 있지만 멋대로 옮기거나 할 수 없고 리모트 저장소와 통신하면 자동으로 업데이트된다. 리모트 브랜치는 브랜치 상태를 알려주는 책갈피라고 볼 수 있다. 이 책갈피로 리모트 저장소에서 마지막으로 데이터를 가져온 시점의 상태를 알 수 있다.

리모트 브랜치의 이름은 (remote)/(branch) 형식으로 되어 있다. 예를 들어 리모트 저장소 origin의 master 브랜치를 보고 싶다면 origin/master라는 이름으로 브랜치를 확인하면 된다. 다른 팀원과 함께 어떤 이슈를 구현할 때 그 팀원이 iss53 브랜치를 서버로 Push했고 당신도 로컬에 iss53 브랜치가 있다고 가정하자. 이때 서버가 가리키는 iss53 브랜치는 로컬에서 origin/iss53이 가리키는 커밋이다.

다소 헷갈릴 수 있으니 예제를 좀 더 살펴보자. `git.ourcompany.com`이라는 Git 서버가 있고 이 서버의 저장소를 하나 Clone하면 Git은 자동으로 origin이라는 이름을 붙인다. origin으로부터 저장소 데이터를 모두 내려받고 master 브랜치를 가리키는 포인터를 만든다. 이 포인터는 origin/master라고 부르고 멋대로 조종할 수 없다. 그리고 Git은 로컬의 master 브랜치가 origin/master를 가리키게 한다. 이제 이 master 브랜치에서 작업을 시작할 수 있다.

로컬 저장소에서 어떤 작업을 하고 있는데 동시에 다른 팀원이 `git.ourcompany.com` 서버에 Push하고 master 브랜치를 업데이트한다. 그러면 이제 팀원 간의 히스토리는 서로 달라진다. 서버 저장소로부터 어떤 데이터도 주고받지 않아서 origin/master 포인터는 그대로다.

리모트 서버로부터 저장소 정보를 동기화하려면 `git fetch origin` 명령을 사용한다. 명령을 실행하면 우선 origin 서버의 주소 정보(이 예에서는 `git.ourcompany.com`)를 찾아서, 현재 로컬의 저장소가 갖고 있지 않은 새로운 정보가 있으면 모두 내려받고, 받은 데이터를 로컬 저장소에 업데이트하고 나서,

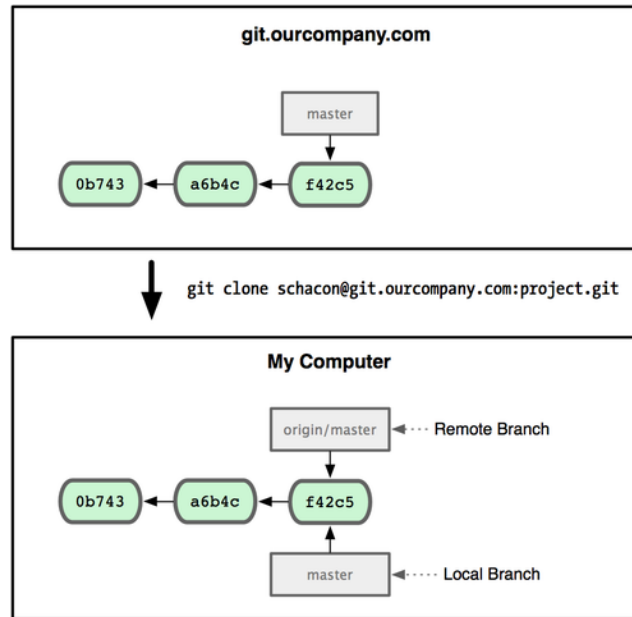



그림 3.22: 저장소를 Clone하면 로컬 master 브랜치, 리모트 저장소의 master 브랜치를 가리키는 origin/master 브랜치가 생김

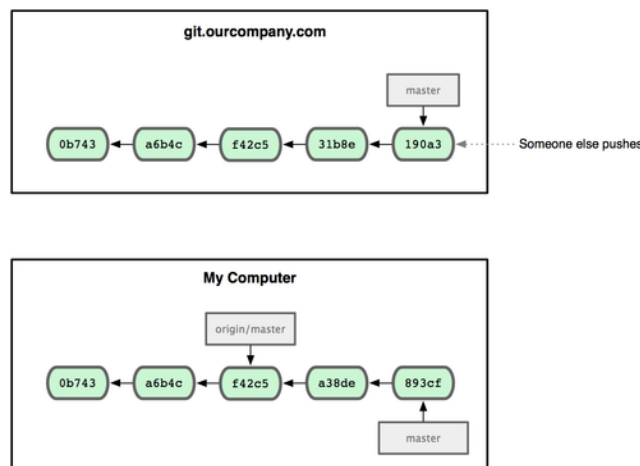



그림 3.23: 로컬과 서버의 커밋 히스토리는 독립적임

origin/master 포인터의 위치를 최신 커밋으로 이동시킨다.

리모트 저장소를 여러 개 운영하는 상황을 이해할 수 있도록 개발용으로 사용할 Git 저장소를 팀 내부에 하나 추가해 보자.

이 저장소의 주소가 `git.team1.ourcompany.com` 이면 2장에서 살펴본 `git remote add` 명령으로 현재 작업 중인 프로젝트에 팀의 저장소를 추가한다. 이름을 teamone으로 짓고 긴 서버 주소 대신 사용한다.

서버를 추가하고 나면 git fetch teamone 명령으로 teamone 서버의 데이터를 내려받는다. 명령을 실행해도 teamone 서버의 데이터는 모두 origin 서버에도 있는 것들이라서 아무것도 내려받지 않는다. 하지만, 이 명령은 teamone/master 브랜치가 teamone 서버의 master 브랜치가 가리키는 커밋을 가리키게 한다.

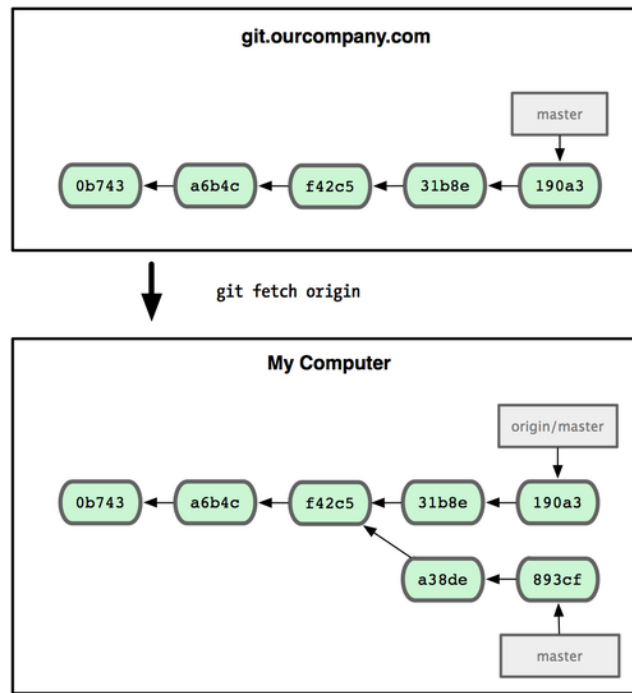



그림 3.24: Git의 Fetch 명령은 리모트 브랜치 정보를 업데이트한다

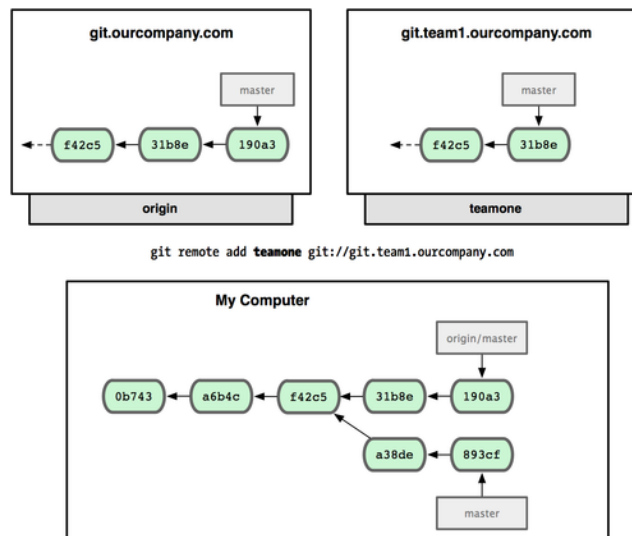



그림 3.25: 서버를 리모트 저장소로 추가하기

3.5.1 Push하기

로컬의 브랜치를 서버로 전송하려면 쓰기 권한이 있는 리모트 저장소에 Push해야 한다. 로컬 저장소의 브랜치는 자동으로 리모트 저장소로 전송되지 않는다. 명시적으로 브랜치를 Push해야 정보가 전송된다. 따라서 리모트 저장소에 전송하지 않고 로컬 브랜치에만 두는 비공개 브랜치를 만들 수 있다. 또 다른 사람과 협업하기 위해 토픽 브랜치만 전송할 수도 있다.

serverfix라는 브랜치를 다른 사람과 공유할 때에도 브랜치를 처음 Push하는 것과 같은 방법으로 Push한다. 다음과 같이 `git push (remote) (branch)` 명령을 사용한다:

```
$ git push origin serverfix
```

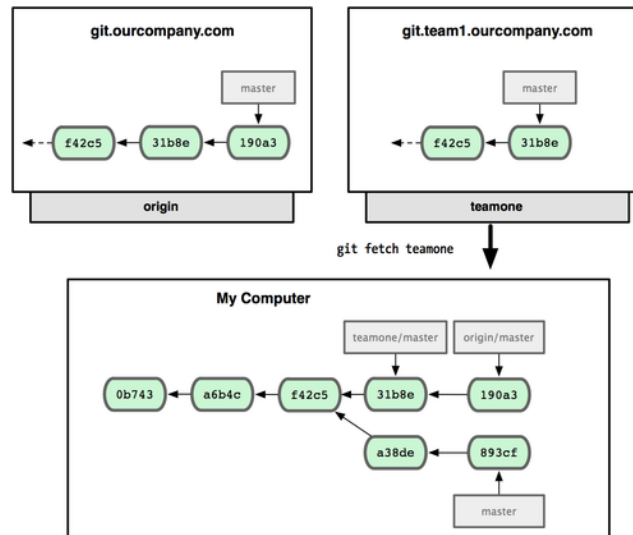



그림 3.26: 로컬 저장소에 만들어진 teamone의 master 브랜치를 가리키는 포인터

```
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

이 메시지에는 숨겨진 내용이 많다.

Git은 serverfix라는 브랜치 이름을 `refs/heads/serverfix:refs/heads/serverfix`로 확장한다. 이것은 serverfix라는 로컬 브랜치를 서버로 Push하는데 리모트의 serverfix 브랜치로 업데이트한다는 것을 의미한다. 나중에 9장에서 `refs/heads/`의 뜻을 자세히 알아볼 것이기 때문에 일단 넘어가도록 한다. `git push origin serverfix:serverfix`라고 Push하는 것도 같은 의미인데 이것은 '로컬의 serverfix 브랜치를 리모트 저장소의 serverfix 브랜치로 Push하라'라는 뜻이다. 로컬 브랜치의 이름과 리모트 서버의 브랜치 이름이 다를 때 필요하다. 리모트 저장소에 serverfix라는 이름 대신 다른 이름을 사용하려면 `git push origin serverfix:awesomebranch`처럼 사용한다.

나중에 누군가 저장소를 Fetch하고 나서 서버에 있는 serverfix 브랜치에 접근할 때 origin/serverfix 라는 이름으로 접근할 수 있다:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

여기서 짚고 넘어가야 할 게 있다. Fetch 명령으로 리모트 브랜치를 내려받는다고 해서 로컬 저장소에

수정할 수 있는 브랜치가 새로 생기는 것이 아니다. 다시 말해서 serverfix라는 브랜치가 생기는 것이 아니라 그저 수정 못 하는 origin/serverfix 브랜치 포인터가 생기는 것이다.

새로 받은 브랜치의 내용을 Merge하려면 `git merge origin/serverfix` 명령을 사용한다. Merge하지 않고 리모트 브랜치에서 시작하는 새 브랜치를 만들려면 아래와 같은 명령을 사용한다.

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

그러면 origin/serverfix에서 시작하고 수정할 수 있는 serverfix라는 로컬 브랜치가 만들어진다.

3.5.2 브랜치 추적

리모트 브랜치를 로컬 브랜치로 Checkout하면 자동으로 트래킹(Tracking) 브랜치가 만들어진다. 트래킹 브랜치는 리모트 브랜치와 직접적인 연결고리가 있는 로컬 브랜치이다. 트래킹 브랜치에서 `git push` 명령을 내려도 Git은 연결고리가 있어서 어떤 리모트 저장소에 Push해야 하는지 알 수 있다. 또한 `git pull` 명령을 내리면 리모트 저장소로부터 데이터를 내려받아 연결된 리모트 브랜치와 자동으로 Merge한다.

서버로부터 저장소를 Clone해올 때도 Git은 자동으로 master 브랜치를 origin/master 브랜치의 트래킹 브랜치로 만든다. 그래서 `git push`, `git pull` 명령이 추가적인 아규먼트 없이도 동작한다. 트래킹 브랜치를 직접 만들 수 있는데 origin/master뿐만 아니라 다른 저장소의 다른 브랜치도 추적하게(Tracking) 할 수 있다. `git checkout -b [branch] [remotename]/[branch]` 명령으로 간단히 트래킹 브랜치를 만들 수 있다. Git 1.6.2 버전 이상을 사용하는 경우에는 - track 옵션도 사용할 수 있다.

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch serverfix from origin.
Switched to a new branch 'serverfix'
```

리모트 브랜치와 다른 이름으로 브랜치를 만들려면 로컬 브랜치의 이름을 아래와 같이 다르게 지정한다:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch serverfix from origin.
Switched to a new branch 'sf'
```

이제 `sf` 브랜치에서 Push나 Pull하면 자동으로 `origin/serverfix`에 데이터를 보내거나 가져온다.

3.5.3 리모트 브랜치 삭제

동료와 협업하기 위해 리모트 브랜치를 만들었다가 작업을 마치고 master 브랜치로 Merge했다. 협업하는 데 사용했던 그 리모트 브랜치는 이제 안정화됐으므로 삭제할 수 있다. `git push [remotename] :[branch]`라고 실행해서 삭제할 수 있는데 이 명령은 좀 특이하게 생겼다. serverfix라는 리모트 브랜치를 삭제하려면 다음과 같이 실행한다:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

위 명령을 실행하고 나면 서버의 브랜치는 삭제된다. 이 명령을 잊어버릴 경우를 대비해서 페이지 귀퉁이를 접어놓고 필요할 때 펴보는 게 좋을지도 모르겠다. 이 명령은 앞서 살펴본 `git push [remotename] [localbranch]:[remotebranch]` 형식으로 기억하는 것이 좋다. `[localbranch]` 부분에 비워 둔 채로 실행하면 '로컬에서 빈 내용을 리모트의 `[remotebranch]`에 채워 넣어라' 라는 뜻이 되기 때문이다.

3.6 Rebase하기

Git에서 한 브랜치에서 다른 브랜치로 합치는 방법은 두 가지가 있다. 하나는 Merge이고 다른 하나는 Rebase다. 이 절에서는 Rebase가 무엇인지, 어떻게 사용하는지, 좋은 점은 뭐고, 어떤 상황에서 사용하고 어떤 상황에서 사용하지 말아야 하는지 알아 본다.

3.6.1 Rebase의 기초

앞의 Merge 절에서 살펴본 예제로 다시 돌아가 보자(그림 3-27). 두 개의 나누어진 브랜치의 모습을 볼 수 있다.

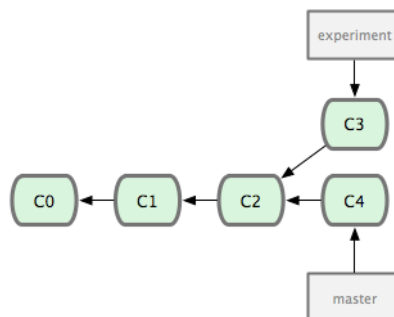



그림 3.27: 두 개의 브랜치로 나누어진 커밋 히스토리

이 두 브랜치를 합치는 가장 쉬운 방법은 앞에서 살펴본 대로 Merge 명령을 사용하는 것이다. 두 브랜치의 마지막 커밋 두 개(C3, C4)와 공통 조상(C2)을 사용하는 3-way Merge로 그림 3-28처럼 새로운 커밋을 만들어 낸다.

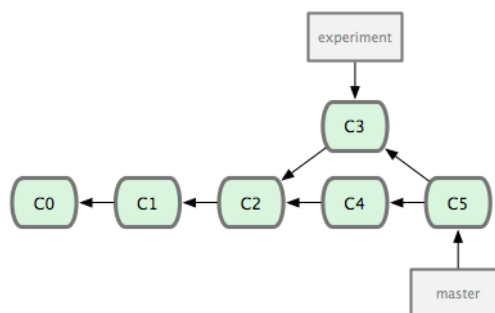



그림 3.28: 나뉜 브랜치를 Merge하기

비슷한 결과를 만드는 다른 방식으로, C3에서 변경된 사항을 패치(Patch)로 만들고 이를 다시 C4에 적용시키는 방법이 있다. Git에서는 이런 방식을 Rebase 라고 한다. Rebase 명령으로 한 브랜치에서 변경된 사항을 다른 브랜치에 적용할 수 있다.
위의 예제는 다음과 같은 명령으로 Rebase한다:

```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

실제로 일어나는 일을 설명하자면 일단 두 브랜치가 나뉘기 전인 공통 커밋으로 이동하고 나서 그 커밋부터 지금 Checkout한 브랜치가 가리키는 커밋까지 diff를 차례로 만들어 어딘가에 임시로 저장해 놓는다. Rebase할 브랜치(역주 - experiment)가 합칠 브랜치(역주 - master)가 가리키는 커밋을 가리키게 하고 아까 저장해 놓았던 변경사항을 차례대로 적용한다. 그림 3-29는 이러한 과정을 나타내고 있다.

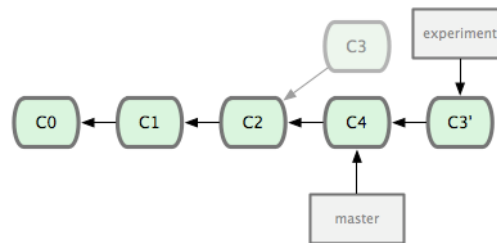



그림 3.29: C3의 변경사항을 C4에 적용하는 Rebase 과정

그리고 나서 master 브랜치를 Fast-forward 시킨다.

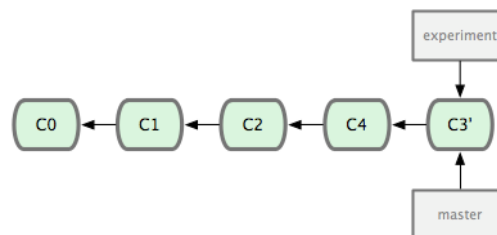



그림 3.30: master 브랜치를 Fast-forward시키기

C3'로 표시된 커밋에서의 내용은 Merge 예제에서 살펴본 C5 커밋에서의 내용과 같을 것이다. Merge이든 Rebase든 둘 다 합치는 관점에서는 서로 다를 게 없다. 하지만, Rebase가 좀 더 깨끗한 히스토리를 만든다. Rebase한 브랜치의 Log를 살펴보면 히스토리가 선형적이다. 일을 병렬로 동시에 진행해도 Rebase하고 나면 모든 작업이 차례대로 수행된 것처럼 보인다.
Rebase는 보통 리모트 브랜치에 커밋을 깔끔하게 적용하고 싶을 때 사용한다. 아마 이렇게 Rebase하는 리모트 브랜치는 직접 관리하는 것이 아니라 그냥 참여하는 브랜치일 것이다. 메인 프로젝트에 패치를 보낼 준비가 되면 하는 것이 Rebase이니까 브랜치에서 하던 일을 완전히 마치고 origin/master로 Rebase한다. 프로젝트 관리자는 어떠한 통합작업도 필요 없다. 그냥 master 브랜치를 Fast-forward 시키면 된다.
Rebase를 하든지, Merge를 하든지 최종 결과물은 같고 커밋 히스토리만 다르다는 것이 중요하다. Rebase의 경우는 브랜치의 변경사항을 순서대로 다른 브랜치에 적용하면서 합치고 Merge의 경우는

두 브랜치의 최종결과만을 가지고 합친다.

3.6.2 좀 더 Rebase

Rebase는 단순히 브랜치를 합치는 것만 아니라 다른 용도로도 사용할 수 있다. 그림 3-31과 같은 히스토리가 있다고 하자. server 브랜치를 만들어서 서버 기능을 추가하고 그 브랜치에서 다시 client 브랜치를 만들어 클라이언트 기능을 추가한다. 마지막으로 server 브랜치로 돌아가서 몇 가지 기능을 더 추가한다.

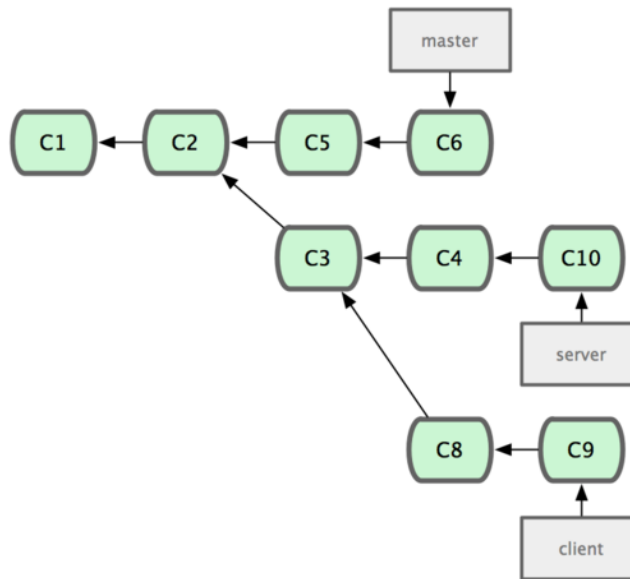



그림 3.31: 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치

이때 테스트가 덜 된 server 브랜치는 그대로 두고 client 브랜치만 master로 합치려는 상황을 생각해보자. server와는 아무 관련이 없는 client 커밋은 C8, C9이다. 이 두 커밋을 master 브랜치에 적용하기 위해서 `--onto` 옵션을 사용하여 아래와 같은 명령을 실행한다:

```
$ git rebase --onto master server client
```

이 명령은 client 브랜치를 Checkout하고 server와 client의 공통조상 이후의 패치를 만들어 master에 적용한다. 조금 복잡하긴 해도 꽤 쓸모 있다. 그림 3-32를 보자.
이제 master 브랜치로 돌아가서 Fast-forward 시킬 수 있다:

```
$ git checkout master
$ git merge client
```

server 브랜치의 일이 다 끝나면 `git rebase [basebranch] [topicbranch]`라는 명령으로 Checkout하지 않고 바로 server 브랜치를 master 브랜치로 rebase할 수 있다. 이 명령은 토픽(server) 브랜치를 Checkout하고 베이스(master) 브랜치에 Rebase한다:

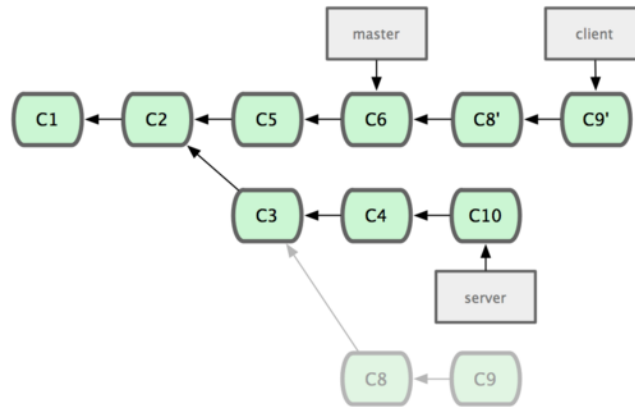



그림 3.32: 다른 토픽 브랜치에서 갈라져 나온 토픽 브랜치를 Rebase하기

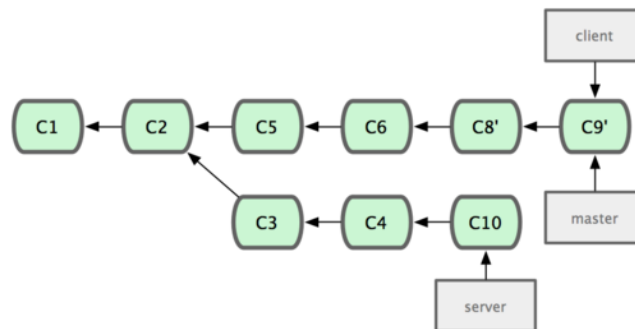



그림 3.33: master 브랜치를 client 브랜치 위치로 진행 시키기

```
$ git rebase master server
```

server 브랜치의 수정사항을 master 브랜치에 적용했다. 그 결과는 그림 3-34와 같다.

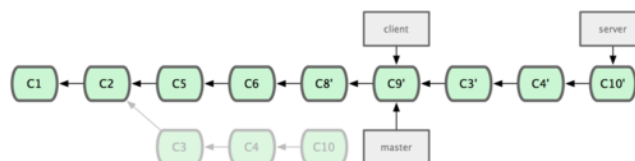



그림 3.34: master 브랜치에 server 브랜치의 수정 사항을 적용

그리고 나서 master 브랜치를 Fast-forward 시킨다:

```
$ git checkout master
$ git merge server
```

모든 것이 master 브랜치에 통합됐기 때문에 더 필요하지 않다면 client나 server 브랜치는 삭제해도 된다. 브랜치를 삭제해도 커밋 히스토리는 그림 3-35와 같이 여전히 남아 있다:

```
$ git branch -d client
$ git branch -d server
```

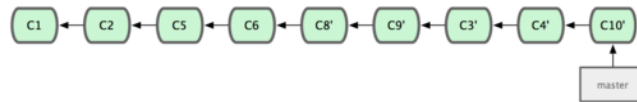



그림 3.35: 최종 커밋 히스토리

3.6.3 Rebase의 위험성

Rebase가 장점이 많은 기능이지만 단점이 없는 것은 아니니 조심해야 한다. 그 주의사항은 다음 한 문장으로 표현할 수 있다:

이미 공개 저장소에 Push한 커밋을 Rebase하지 마라

이 지침만 지키면 Rebase를 하는 데 문제 될 게 없다. 하지만, 이 주의사항을 지키지 않으면 사람들에게 욕을 먹을 것이다(역주 - 아마도 가카의 호연지기가 필요해질 것이다).

Rebase는 기존의 커밋을 그대로 사용하는 것이 아니라 내용은 같지만 다른 커밋을 새로 만든다. 새 커밋을 서버에 Push하고 동료 중 누군가가 그 커밋을 Pull해서 작업을 한다고 하자. 그런데 그 커밋을 `git rebase`로 바꿔서 Push해버리면 동료가 다시 Push했을 때 동료는 다시 Merge해야 한다. 그리고 동료가 다시 Merge한 내용을 Pull하면 내 코드는 정말 엉망이 된다.

이미 공개 저장소에 Push한 커밋을 Rebase하면 어떤 결과가 초래되는지 예제를 통해 알아보자. 중앙 저장소에서 Clone하고 일부 수정을 하면 커밋 히스토리는 그림 3-36과 같아 진다.

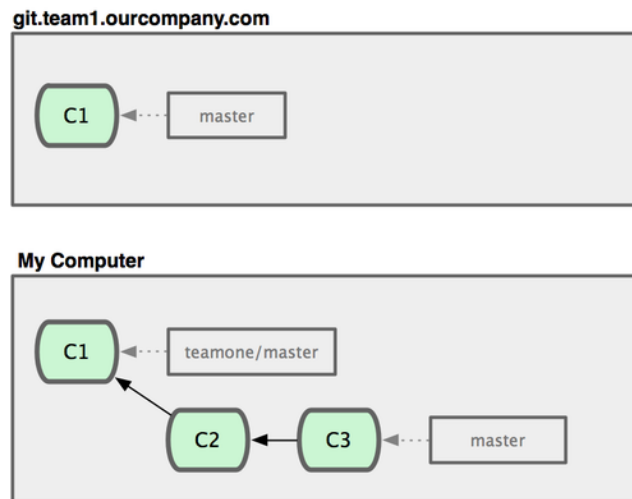



그림 3.36: 저장소를 Clone하고 일부 수정함

이제 팀원 중 누군가 커밋, Merge하고 나서 서버에 Push 한다. 이 리모트 브랜치를 Fetch, Merge하면 그림 3-37과 같이 된다.

그런데 Push했던 팀원은 Merge한 일을 되돌리고 다시 Rebase한다. 서버의 히스토리를 새로 덮어씌우려면 `git push --force` 명령을 사용해야 한다. 이후에 저장소에서 Fetch하고 나면 아래 그림과 같은 상태가 된다:

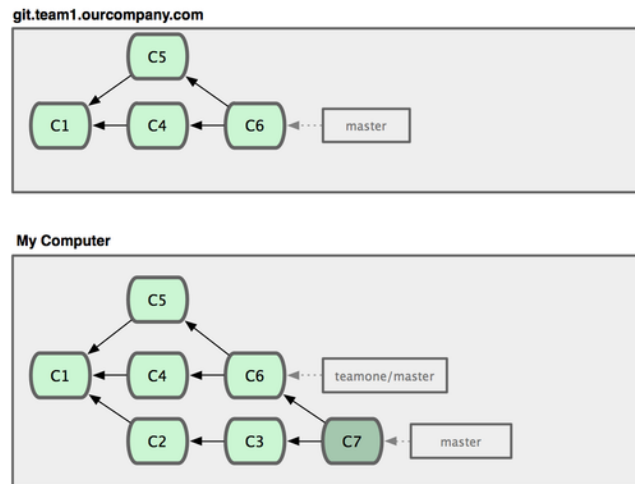



그림 3.37: Fetch한 후 Merge함

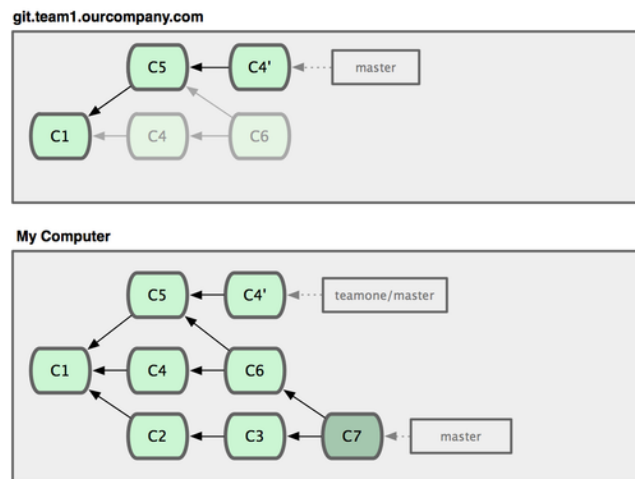



그림 3.38: 한 팀원이 다른 팀원이 의존하는 커밋을 없애고 Rebase한 커밋을 다시 Push함

기존 커밋이 사라졌기 때문에 이미 처리한 일이라고 해도 다시 Merge해야 한다. Rebase는 커밋의 SHA-1 해시를 바꾸기 때문에 Git은 새로운 커밋으로 생각한다. 사실 C4는 이미 히스토리에 적용되어 있지만, Git은 모른다.

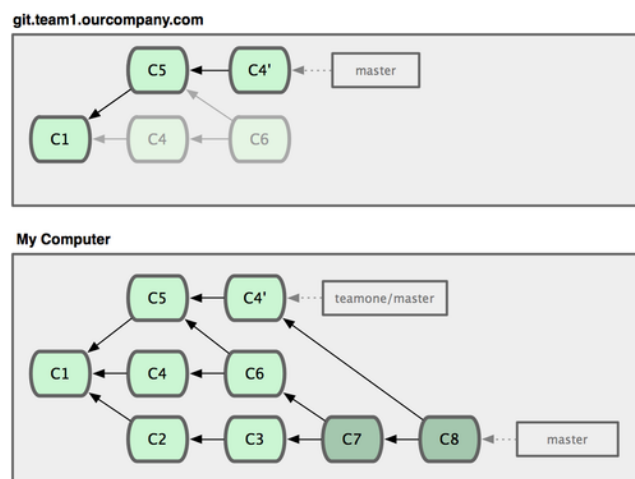



그림 3.39: 같은 Merge를 다시 한다

다른 개발자와 계속 같이 일하려면 이런 Merge도 해야만 한다. Merge하면 C4와 C4' 커밋 둘 다 히스토리에 남게 된다. 실제 내용과 메시지가 같지만 SHA-1 해시 값이 전혀 다르다. `git log`로 히스토리를 확인해보면 저자, 커밋 날짜, 메시지가 같은 커밋이 두 개 있을 것이다. 이렇게 되면 혼란스럽다. 게다가 이 히스토리를 서버에 Push하면 같은 커밋이 두 개 있기 때문에 다른 사람들도 혼란스러워한다.

Push하기 전에 정리하려고 Rebase하는 것은 괜찮다. 또 절대 공개하지 않고 혼자 Rebase하는 경우도 괜찮다. 하지만, 이미 공개하여 사람들이 사용하는 커밋을 Rebase하면 틀림없이 문제가 생길 것이다.

3.7 요약

우리는 이 장에서 Git으로 브랜치를 만들고 Merge 기능의 기본적인 명령을 다루었다. 이제 브랜치를 만들고 옮겨다니고 Merge하는 것에 익숙해졌을 것으로 생각한다. 브랜치를 Push하여 공유하거나 Push하기 전에 브랜치를 Rebase하는 것 정도는 어렵지 않게 할 수 있을 것이다.

4장

Git 서버

이 글을 읽는 독자라면 이미 하루 업무의 대부분을 Git으로 처리할 수 있을 거라고 생각한다. 이제는 다른 사람과 협업하는 방법을 고민해보자. 다른 사람과 협업하려면 리모트 저장소가 필요하다. 물론 혼자서 저장소를 만들고 거기에 Push하고 Pull할 수도 있지만 이렇게 하는 것은 아무 의미가 없다. 이런 방식으로는 다른 사람이 무슨 일을 하고 있는지 알려면 항상 지켜보고 있어야 간신히 알 수 있을 터이다. 당신 컴퓨터가 오프라인일 때에도 동료가 저장소를 사용할 수 있도록 언제나 이용할 수 있는 저장소가 필요하다. 즉, 공동으로 사용할 수 있는 저장소를 만들고 모두 이 저장소에 접근하여 Push, Pull할 수 있어야 한다. 우리는 이 저장소를 “Git 서버”라고 부른다. Git 저장소를 운영하는데 자원이 많이 필요하지도 않아서 별도로 Git 서버를 준비하지 않아도 된다.

Git 서버를 운영하는 것은 어렵지 않다. 우선 사용할 전송 프로토콜부터 정한다. 이 장의 앞부분에서는 어떤 프로토콜이 있는지 그리고 각 장단점은 무엇인지 살펴본다. 그다음엔 각 프로토콜을 사용하는 방법과 그 프로토콜을 사용할 수 있도록 서버를 구성하는 방법을 살펴본다. 마지막으로 다른 사람의 서버에 내 코드를 맡기긴 싫고 고생스럽게 서버를 설치하고 관리하고 싶지도 않을 때 고를 수 있는 선택지가 어떤 것들이 있는지 살펴본다.

서버를 직접 설치해서 운영할 생각이 없으면 이 장의 마지막 절만 읽어도 된다. 마지막 절에서는 Git 호스팅 서비스에 계정을 만들고 사용하는 방법에 대해 설명한다. 그리고 다음 장에서는 분산 환경에서 소스를 관리하는 다양한 패턴에 대해 논의할 것이다.

리모트 저장소는 일반적으로 워킹 디렉토리가 없는 Bare 저장소 이다. 이 저장소는 협업용이기 때문에 체크아웃이 필요 없다. 그냥 Git 데이터만 있으면 된다. 다시 말해서 Bare 저장소는 일반 프로젝트에서 `.git` 디렉토리만 있는 저장소다.

4.1 프로토콜

Git은 Local, SSH, Git, HTTP 이렇게 네 가지의 네트워크 프로토콜을 사용할 수 있다. 이 절에서는 각각 어떤 경우에 유용한지 살펴볼 것이다.

HTTP 프로토콜을 제외한 나머지들은 모두 Git이 서버에 설치돼 있어야 한다.

4.1.1 로컬 프로토콜

가장 기본적인 것이 로컬 프로토콜 이다. 리모트 저장소가 단순히 디스크의 다른 디렉토리에 있을 때 사용한다. 팀원들이 전부 한 시스템에 로그인하여 개발하거나 아니면 NFS같은 것으로 파일시스템을 공유하고 있을 때 사용한다. 전자는 문제가 될 수 있다. 모든 저장소가 한 시스템에 있기 때문에 한순간에 모두 잃을 수 있다.

공유 파일시스템을 마운트했을 때는 로컬 저장소를 사용하는 것처럼 Clone하고 Push하고 Pull하면 된다. 일단 저장소를 Clone하거나 프로젝트에 리모트 저장소로 추가한다. 추가할 때 URL 자리에 저장소의 경로를 사용한다. 예를 들어 아래와 같이 로컬 저장소를 Clone한다:

```
$ git clone /opt/git/project.git
```

아래 처럼도 가능하다:

```
$ git clone file:///opt/git/project.git
```

Git은 파일 경로를 직접 쓸 때와 `file://`로 시작하는 URL을 사용할 때에 약간 다르게 처리한다. 디렉토리 경로를 사용해서 같은 파일시스템에 있는 저장소를 Clone할 때 Git은 하드링크를 만든다. 같은 파일시스템에 있는게 아니면 그냥 복사한다. 하지만 `file://`로 시작하면 Git은 네트워크를 통해서 데이터를 전송할 때처럼 프로세스를 별도로 생성하여 처리한다. 이 프로세스로 데이터를 전송하는 것은 효율이 좀 떨어지지만 그래도 `file://`를 사용하는 이유가 있다. 보통은 다른 버전 관리 시스템들에서 임포트한 후에 이렇게 사용하는데, 외부 레퍼런스나 개체들이 포함된 저장소의 복사본을 깨끗한 상태로 남겨두고자 할때 사용한다(9장에서 자세히 다룬다). 여기서는 속도가 빠른 디렉토리 경로를 사용한다.
이미 있는 Git 프로젝트에서 아래와 같이 로컬 저장소를 추가한다:

```
$ git remote add local_proj /opt/git/project.git
```

그러면 네트워크에 있는 리모트 저장소처럼 Push하고 Pull할 수 있다.

장점

파일 기반 저장소는 단순한 것이 장점이다. 기존에 있던 네트워크나 파일의 권한을 그대로 사용하기 때문에 설정하기 쉽다. 이미 팀 전체가 접근 가능한 파일시스템이 있으면 저장소를 아주 쉽게 구성할 수 있다. 디렉토리를 공유하듯이 동료가 모두 읽고 쓸 수 있는 공유 디렉토리에 Bare 저장소를 만든다. 다음 절인 "서버에 Git 설치하기"에서 Bare 저장소를 만드는 방법을 살펴볼 것이다.
또한, 동료가 작업하는 저장소에서 한 일을 바로 가져오기에도 좋다. 만약 함께 프로젝트를 하는 동료가 자신이 한 일을 당신이 확인해 줬으면 한다. 이럴 때 그 동료가 서버에 Push하고 당신이 다시 Pull할 필요없이 `git pull /home/john/project` 라는 명령어를 바로 실행시켜서 매우 쉽게 동료의 코드를 가져올 수 있다.

단점

다양한 상황에서 접근할 수 있도록 디렉토리를 공유하는 것 자체가 일반적으로 어렵다. 집에 있을 때 Push하려면 리모트 저장소가 있는 디스크를 마운트해야 하는데 이것은 다른 프로토콜을 이용하는 방법보다 느리고 어렵다.
게다가 네트워크 파일시스템을 마운트해서 사용하는 중이라면 별로 빠르지도 않다. 로컬 저장소는 데이터를 빠르게 읽을 수 있을 때만 빠르다. NFS에 있는 저장소에 Git을 사용하는 것은 보통 같은 서버에 SSH로 접근하는 것보다 느리다.

4.1.2 SSH 프로토콜

Git의 대표 프로토콜은 SSH이다. 대부분 서버는 SSH로 접근할 수 있도록 설정돼 있다. 뭐, 설정돼 있지 않더라도 쉽게 설정할 수 있다. 그리고 SSH는 읽기/쓰기 접근을 쉽게 할 수 있는 유일한 네트워크 프로토콜이다. 다른 네트워크 프로토콜인 HTTP와 Git은 일반적으로 읽기만 가능하다. 그래서 초보자(unwashed masses)라고 해도 쓰기 명령을 이용하려면 SSH가 필요하다. SSH는 또한 인증도 지원한다. SSH는 보통 유비쿼터스 적이면서도, 사용하기도, 설치하기도 쉽다.

SSH를 통해 Git 저장소를 Clone하려면 `ssh://`로 시작하는 URL을 사용한다:

```
$ git clone ssh://user@server/project.git
```

아니면 scp 명령어처럼 사용할 수 있다. 이게 조금 더 짧다:

```
$ git clone user@server:project.git
```

사용자 계정을 생략할 수도 있는데 계정을 생략하면 Git은 현재 로그인한 사용자의 계정을 사용한다.

장점

SSH는 장점이 매우 많은 프로토콜이다. 첫째, 누가 리모트에서 저장소에 접근하는지 알고 싶다면 SSH를 사용해야 한다. 둘째, SSH는 상대적으로 설정하기 쉽다. SSH 데몬은 정말 흔하다. 네트워크 관리자은 SSH 데몬을 다루어본 경험이 있고 대부분의 OS 배포판에는 SSH 데몬과 관리도구가 모두 들어 있다. 셋째, SSH를 통해 접근하면 보안에 안전하다. 모든 데이터는 암호화되어 인증된 상태로 전송된다. 마지막으로 SSH는 전송 시 데이터를 가능한 압축하기 때문에 효율적이다.

단점

SSH의 단점은 익명으로 접근할 수 없다는 것이다. 심지어 읽기 전용인 경우에도 익명으로 시스템에 접근할 수 없다. 회사에서만 사용할 것이라면 SSH가 가장 적합한 프로토콜일 것이지만 오픈소스 프로젝트는 SSH만으로는 부족하다. 만약 사람들이 프로젝트에 익명으로 접근할 수 있게 하려면, 자신이 Push할 때 사용할 SSH를 설치하는 것과 별개로 다른 사람들이 Pull할 때 사용할 다른 프로토콜을 추가해야 한다.

4.1.3 Git 프로토콜

Git 프로토콜은 Git에 포함된 데몬을 사용하는 방법이다. 포트는 9418이며 SSH 프로토콜과 비슷한 서비스를 제공하지만, 인증 메커니즘이 없다. 저장소에 git-export-daemon-ok 파일을 만들면 Git 프로토콜로 서비스할 수 있지만, 보안은 없다. 이 파일이 없는 저장소는 Git 프로토콜로 서비스할 수 없다. 이 저장소는 누구나 Clone할 수 있거나 아무도 Clone할 수 없거나 둘 중의 하나만 선택할 수 있다. 그래서 이 프로토콜로는 Push 가능하게 설정할 수 없다. 엄밀히 말해서 Push할 수 있도록 설정할 수 있지만, 인증하도록 할 수 없다. 그러니까 당신이 Push할 수 있으면 이 프로젝트의 URL을 아는 사람은 누구나 Push할 수 있다. 그냥 이런 것도 있지만 잘 안 쓴다고 알고 있으면 된다.

장점

Git 프로토콜은 전송속도가 가장 빠르다. 전송량이 많은 공개 프로젝트나 별도의 인증이 필요 없고 읽기만 허용하는 프로젝트를 서비스할 때 유용하다. 암호화와 인증을 빼면 SSH 프로토콜과 전송 메커니즘이 별반 다르지 않다.

단점

Git 프로토콜은 인증 메커니즘이 없는게 단점이다. Git 프로토콜만 사용하는 프로젝트는 바람직하지 못하다. 일반적으로 SSH 프로토콜과 함께 사용한다. 소수의 개발자만 Push할 수 있고 대다수 사람은 `git://`을 사용하여 읽을 수만 있게 한다. 어쩌면 가장 설치하기 어려운 방법일 수도 있다. 별도의 데몬이 필요하고 프로젝트에 맞게 설정해야 한다. 이 장의 Gitosis 절에서 설정하는 법을 살펴볼 것이다. 자원을 아낄 수 있도록 xinetd 같은 것도 설정해야 하고 방화벽을 통과할 수 있도록 9418 포트도 열어야 한다. 이 포트는 일반적으로 회사들이 허용하는 표준 포트가 아니다. 규모가 큰 회사라면 당연히 방화벽에서 이 포트를 막아 놓는다.

4.1.4 HTTP/S 프로토콜

마지막으로, HTTP 프로토콜이 있다. HTTP와 HTTPS 프로토콜의 미학은 설정이 간단하다는 점이다. HTTP 도큐먼트 루트 밑에 Bare 저장소를 두고 post-update 훅을 설정하는 것이 기본적으로 해야 하는 일의 전부다(7장에서 Git 훅에 대해 자세히 다룰 것이다). 저장소가 있는 웹 서버에 접근할 수 있다면 그 저장소를 Clone할 수도 있다. HTTP를 통해서 저장소를 읽을 수 있게 하려면 아래와 같이 한다:

```
$ cd /var/www/htdocs/
$ git clone --bare /path/to/git_project gitproject.git
$ cd gitproject.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

post-update 훅은 Git에 포함되어 있으며 `git update-server-info`라는 명령어를 실행시킨다. 이 명령어는 HTTP로 Fetch와 Clone 명령이 잘 동작하게 한다. SSH를 통해서 저장소에 Push할 때 실행되며, 사람들은 아래와 같이 Clone한다:

```
$ git clone http://example.com/gitproject.git
```

여기서는 Apache 서버가 기본으로 사용하는 `/var/www/htdocs`을 루트 디렉토리로 사용하지만 다른 웹 서버를 사용해도 된다. 단순히 Bare 저장소를 HTTP 문서 루트에 넣으면 된다. Git 데이터는 일반적인 정적 파일처럼 취급된다(9장에서 정확히 어떻게 처리하는지 다룰 것이다).
HTTP를 통해서 Push하는 것도 가능하다. 단지 이 방법은 잘 사용하지 않는 WebDAV 환경을 완벽하게 구축해야 한다. 잘 사용하지 않기 때문에 이 책에서도 다루지 않는다. HTTP 프로토콜로 Push하고 싶으면 `http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt` 읽고 저장소를 만들면 된다. HTTP를 통해서 Push하는 방법의 좋은 점은 WebDAV 서버를 아무거나 골라 쓸 수 있다는 것이다. 그래서 WebDAV를 지원하는 웹 호스팅 업체를 이용하면 이 기능을 사용할 수 있다.

장점

HTTP 프로토콜은 설정하기 쉽다는 것이 장점이다. 몇 개의 필수 명령어만 실행하면 세계 어디에서나 당신의 저장소에 접근할 수 있게 만들 수 있다. 이렇게 하는데 몇 분이면 충분하다. HTTP 프로토콜은 서버의 리소스를 많이 잡아먹지도 않는다. 보통은 정적 HTTP 서버만으로도 충분하기 때문에 흔한 Apache 서버로 초당 수천 개의 파일을 처리할 수 있다. 작은 서버로도 충분히 감당할 수 있다.

또 HTTPS를 사용해서 서비스할 수도 있기 때문에 전송하는 데이터를 암호화할 수 있다. 그리고 클라이언트가 서명된 SSL 인증서를 사용하게 할 수도 있다. 이렇게 하더라도 SSH 공개키를 사용하는 방식보다 쉽다. 서명한 SSL 인증서를 사용하는 게 나을 때도 있고 단순히 HTTPS위에서 HTTP기반 인증을 사용하는 게 나을 때도 있다.

HTTP는 매우 보편적인 프로토콜이라서 거의 모든 회사가 트래픽이 방화벽을 통과하도록 허용한다는 장점도 있다.

단점

클라이언트에서는 HTTP가 좀 비효율적이다. 저장소에서 Fetch하거나 Clone할 때 좀 더 오래 걸린다. 다른 프로토콜의 네트워크 오버헤드보다 HTTP의 오버헤드가 좀 더 크다. 지능적으로 정말 필요한 데이터만 전송하지 않기 때문에 HTTP 프로토콜은 멍청한 프로토콜(Dumb Protocol)이라고도 부른다. 효율적으로 전송하고자 서버는 아무것도 하지 않는다. HTTP와 다른 프로토콜의 성능 차이는 9장에서 자세히 설명한다.

4.2 서버에 Git 설치하기

어떤 서버를 설치하더라도 일단 저장소를 Bare 저장소로 만들어야 한다. 다시 말하지만, Bare 저장소는 워킹 디렉토리가 없는 저장소이다. `--bare` 옵션을 주고 Clone하면 새로운 Bare 저장소가 만들어진다. Bare 저장소 디렉토리는 관례에 따라. git 확장자로 끝난다:

```
$ git clone --bare my_project my_project.git
Cloning into bare repository 'my_project.git'...
done.
```

이 명령이 출력하는 메시지가 조금 이상해보일 수도 있다. 사실 `git clone` 명령은 `git init`을 하고 나서 `git fetch`를 실행한다. 그런데 빈 디렉토리밖에 만들지 않는 `git init` 명령의 메시지만 보여준다. 개체 전송에 관련된 메시지는 아무것도 보여주지 않는다. 전송 메시지를 보여주지 않지만 `my_project.git` 디렉토리를 보면 Git 데이터가 들어 있다.

아래와 같이 실행한 것과 비슷하다:

```
$ cp -Rf my_project/.git my_project.git
```

물론 설정 상의 미세한 차이가 있지만, 저장소의 내용만 고려한다면 같다고 볼 수 있다. 워킹 디렉토리가 없는 Git 저장소인 데다가 별도의 디렉토리도 하나 만들었다는 점에서는 같다.

4.2.1 서버에 Bare 저장소 넣기

Bare 저장소는 이제 만들었으니까 서버에 넣고 프로토콜을 설정한다. `git.example.com`라는 이름의 서버를 하나 준비하자. 그리고 그 서버에 SSH로 접속할 수 있게 하고 `/opt/git`에 Git 저장소를 만든다. 아래와 같이 Bare 저장소를 복사한다:

```
$ scp -r my_project.git user@git.example.com:/opt/git
```

이제 다른 사용자들은 SSH로 서버에 접근해서 저장소를 Clone할 수 있다. 사용자는 `/opt/git` 디렉토리에 읽기 권한이 있어야 한다:

```
$ git clone user@git.example.com:/opt/git/my_project.git
```

이 서버에 SSH로 접근할 수 있는 사용자가 `/opt/git/my_project.git` 디렉토리에 쓰기 권한까지 가지고 있으면 바로 Push할 수 있다. `git init` 명령에 `--shared` 옵션을 추가하면 Git은 자동으로 그룹 쓰기 권한을 추가한다:

```
$ ssh user@git.example.com
$ cd /opt/git/my_project.git
$ git init --bare --shared
```

Git 저장소를 만드는 것이 얼마나 쉬운지 살펴보았다. Bare 저장소를 만들어 SSH로 접근할 수 있는 서버에 올리면 동료와 함께 일할 준비가 끝난다.

그러니까 Git 서버를 구축하는데 사람이 할 일은 정말 별로 없다. SSH로 접속할 수 있도록 서버에 계정을 만들고 Bare 저장소를 사람들이 읽고 쓸 수 있는 곳에 넣어 두기만 하면 된다. 다른 것은 아무것도 필요 없다.

다음 절에서는 좀 더 정교하게 설정하는 법을 살펴본다. 사용자에게 계정을 만들어 주는 법, 저장소를 읽고 쓸 수 있게 하는 법, Web UI를 설정하는 법, Gitosis를 사용하는 법, 등등은 여기에서 설명하지 않는다. 꼭 기억해야 할 것은 동료와 함께 개발할 때 꼭 필요한 것이 SSH 서버와 Bare 저장소뿐이라는 것이다.

4.2.2 바로 설정하기

만약 창업을 준비하고 있거나 회사에서 Git을 막 도입하려고 할 때처럼 사용할 개발자의 수가 많지 않을 때에는 설정할 게 별로 없다. Git 서버 설정에서 사용자 관리가 가장 골치 아프다. 사람이 많으면 어떤 사용자는 읽기만 가능하게 하고 어떤 사용자는 읽고 쓰기 둘 다 가능하게 하는 것이 좀 까다롭다.

SSH 접근

만약 모든 개발자가 SSH로 접속할 수 있는 서버가 있으면 너무 쉽게 저장소를 만들 수 있다. 앞서 말했듯이 할 일이 별로 없다. 저장소의 권한을 꼼꼼하게 관리해야 하면 그냥 운영체제의 파일시스템 권한관리를 이용한다. 동료가 저장소에 쓰기 접근을 해야 하는 데 아직 SSH로 접속할 수 있는 서버가 없으면

하나 마련해야 한다. 아마 독자에게 서버가 있다면 그 서버에는 이미 SSH 서버가 설치돼 있어서 이미 SSH로 접속하고 있을 것이다.

동료가 접속하도록 하는 방법은 몇 가지가 있다. 첫째로 모두에게 계정을 만들어 주는 방법이 있다. 이 방법이 제일 단순하지만 다소 귀찮은 방법이다. 팀원마다 adduser를 실행시키고 임시 암호를 부여해야 하기 때문에 보통 이 방법을 쓰고 싶어 하지 않는다.

둘째로 서버마다 git이라는 계정을 하나씩 만드는 방법이 있다. 쓰기 권한이 필요한 사용자의 SSH 공개키를 모두 모아서 git 계정의 `~/.ssh/authorized_keys`파일에 모든 키를 입력한다. 그러면 모두 git 계정으로 그 서버에 접속할 수 있다. 이 git 계정은 커밋 데이터에는 아무런 영향을 주지 않는다. 다시 말해서 접속하는 데 사용한 SSH 계정과 커밋에 저장되는 사용자는 아무 상관없다.

이미 LDAP 서버 같은 중앙집중식 인증 소스를 가지고 있으면 SSH 서버가 해당 인증을 이용하도록 할 수도 있다. SSH 인증 메커니즘 중 아무거나 하나 이용할 수 있으면 그 서버에 접속이 가능하다.

4.3 SSH 공개키 만들기

이미 말했듯이 많은 Git 서버들은 SSH 공개키로 인증한다. 공개키를 사용하려면 일단 공개키를 만들어야 한다. 공개키를 만드는 방법은 모든 운영체제가 비슷하다. 먼저 키가 있는지부터 확인하자. 사용자의 SSH 키들은 기본적으로 사용자의 `~/.ssh` 디렉토리에 저장한다. 그래서 만약 디렉토리의 파일을 살펴보면 공개키가 있는지 확인할 수 있다:

```
$ cd ~/.ssh
$ ls
authorized_keys2  id_dsa       known_hosts
config            id_dsa.pub
```

something, something.pub이라는 형식으로 된 파일을 볼 수 있다. something은 보통 `id_dsa`나 `id_rsa`라고 돼 있다. 그중 `.pub`파일이 공개키이고 다른 파일은 개인키이다. 만약 이 파일이 없거나 `.ssh` 디렉토리도 없으면 `ssh-keygen`이라는 프로그램으로 키를 생성해야 한다. `ssh-keygen` 프로그램은 리눅스나 Mac의 SSH 패키지에 포함돼 있고 윈도는 MSysGit 패키지 안에 들어 있다:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

먼저 키를 어디에 저장할지 경로를(`.ssh/id_rsa`) 입력하고 암호를 두 번 입력한다. 이때 암호를 비워두면 키를 사용할 때 암호를 묻지 않는다.

사용자는 그 다음에 자신의 공개키를 Git 서버 관리자에게 보내야 한다. 사용자는 `.pub` 파일의 내용을 복사하여 메일을 보내기만 하면 된다. 공개키는 아래와 같이 생겼다:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```

다양한 운영 체제에서 SSH 키를 만드는 방법이 궁금하면 에 있는 Github 설명서를 찾아보는 게 좋다.

4.4 서버에 설정하기

서버에 설정하는 일을 살펴보자. 일단 Ubuntu같은 표준 리눅스 배포판을 사용한다고 가정한다. 사용자는 아마도 `authorized_keys` 파일로 인증할 것이다. 먼저 `git` 계정을 만들고 사용자 홈 디렉토리에 .ssh 디렉토리를 만든다:

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

`authorized_keys` 파일에 SSH 공개키를 추가해야 사용자가 접근할 수 있다. 추가하기 전에 이미 이메일로 공개키를 몇 개 받아서 가지고 있다고 가정하자. 공개키가 어떻게 생겼는지 다시 한번 확인한다:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCB007n/ww+ouN4gSLKssMxXnBOvf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4oxOj6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyGlLwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
O7TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukEOmfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

`authorized_keys` 파일에 추가한다:

```
$ cat /tmp/id_rsa.john.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.josie.pub >> ~/.ssh/authorized_keys
$ cat /tmp/id_rsa.jessica.pub >> ~/.ssh/authorized_keys
```

`--bare` 옵션을 주고 `git init`을 실행해서 워킹 디렉토리가 없는 빈 저장소를 하나 만든다:

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

이제 John씨, Josie씨, Jessica씨는 이 저장소를 리모트 저장소로 등록하면 브랜치를 Push할 수 있다. 프로젝트마다 적어도 한 명은 서버에 접속하여 Bare 저장소를 만들어야 한다. git 계정과 저장소를 만든 서버의 호스트 이름이 `gitserver`라고 하자. 만약 이 서버가 내부망에 있으면 `gitserver`가 그 서버를 가리키도록 DNS에 설정한다. 그러면 명령을 아래와 같이 사용할 수 있다:

```
# on Johns computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

이제 이 프로젝트를 Clone하고 나서 수정하고 Push한다:

```
$ git clone git@gitserver:/opt/git/project.git
$ cd project
$ vim README
$ git commit -am 'fix for the README file'
$ git push origin master
```

개발자들이 읽고 쓸 수 있는 Git 서버를 간단하게 만들었다.
그리고 `git-shell`이라는 걸 사용해서 보안을 강화할 수 있다. 이 쉘로 git 계정을 사용하는 사용자들이 Git 말고 다른 것을 할 수 없도록 제한하는 것이다. git 계정의 로그인 쉘을 이것으로 설정하면 `git` 사용자는 일반적인 쉘을 사용할 수 없다. 통상의 bash, csh 대신에 `git-shell`을 로그인 쉘로 설정하기만 하면 된다. 이것을 하려면 `/etc/passwd` 파일을 편집한다:

```
$ sudo vim /etc/passwd
```

그리고 아래와 같은 줄을 찾는다:

```
git:x:1000:1000::/home/git:/bin/sh
```

/bin/sh를 /usr/bin/git-shell로(which git-shell 명령으로 어디에 설치됐는지 확인하는게 좋다) 변경한다:

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

이제 git 계정은 Git 저장소에 Push하고 Pull하는 것만 가능하고 서버의 쉘에는 접근할 수 없다. 실제로 로그인을 해보면 아래와 같은 메시지로 로그인이 거절된다:

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

4.5 공개하기

익명의 사용자에게 읽기 접근을 허용하고 싶을 때는 어떻게 해야 할까? 프로젝트를 비공개가 아니라 오픈 소스 프로젝트로 공개한다거나 자동 빌드 서버나 CI(Continuous Integration) 서버가 많아서 계정마다 하나하나 설정해야 할 수 있다. 아니면 그냥 매번 SSH 키를 생성하는 게 귀찮을 수도 있다. 그러니까 그냥 간단하게 익명의 사용자도 읽을 수 있도록 하고 싶을 때는 어떻게 해야 할까?

분명 웹 서버를 설치하는 것이 가장 쉬운 방법이다. 이 장의 첫 부분에 설명했듯이 웹 서버를 설치하고 Git 저장소를 문서 루트 디렉토리에 두고 post-update 훅을 켜기만 하면 된다. 먼저 설명했던 예제를 따라 해보자. /opt/git 디렉토리에 저장소가 있고 서버에 Apache가 설치돼 있다고 가정하자. 아무 웹 서버나 다 사용할 수 있지만, 이 예제에서는 Apache를 사용한다. 여기에서는 이해하는 것이 목적이므로 아주 기본적인 Apache 설정만을 보여줄 것이다.

먼저 이 훅을 설정해야 한다:

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

post-update 훅은 무슨 일을 할까? 기본적으로 다음과 같다:

```
$ cat .git/hooks/post-update
#!/bin/sh
#
# An example hook script to prepare a packed repository for use over
# dumb transports.
#
# To enable this hook, rename this file to "post-update".
#
```

```
exec git-update-server-info
```

SSH를 통해서 서버에 Push하면 Git은 이 명령어를 실행하여 HTTP를 통해서도 Fetch할 수 있도록 파일를 갱신한다.
그다음 Apache 설정에 VirtualHost 항목을 추가한다. 이 항목에서 문서 루트가 Git 저장소의 루트 디렉토리가 되도록 한다. 그리고 `*.gitserver`로 접속하는 사람들이 모두 이 서버에 접속하도록 한다. 와일드카드를 이용하여 VirtualHost항목을 아래와 같이 설정한다:

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
</VirtualHost>
```

그리고 Apache 서버는 `www-data` 권한으로 CGI 스크립트를 실행시키기 때문에 `/opt/git` 디렉토리의 그룹 소유 권한을 `www-data`로 수정해 주어야 웹 서버로 접근하는 사용자들이 읽을 수 있다.

```
$ chgrp -R www-data /opt/git
```

Apache를 재시작하면 아래와 같은 URL로 저장소를 Clone할 수 있다:

```
$ git clone http://git.gitserver/project.git
```

이렇게 사용자들이 HTTP로 프로젝트에 접근하도록 설정하는 데 몇 분밖에 걸리지 않는다. 그리고 Git 데몬으로도 똑같이 인증 없이 접속하게 할 수 있다. 프로세스를 데몬으로 만들어야 한다는 단점이 있지만 가능하다. 이것은 다음 절에서 살펴볼 것이다.

4.6 GitWeb

프로젝트 저장소를 단순히 읽거나 쓰는 것에 대한 설정은 다뤘다. 이제는 웹 기반 인터페이스를 설정해 보자. Git에는 GitWeb이라는 CGI 스크립트를 제공해서 쉽게 웹에서 저장소를 조회하도록 할 수 있다. 같은 사이트에서 GitWeb을 구경할 수 있다(그림 4-1).
Git은 GitWeb을 쉽게 사용해 볼 수 있도록 서버를 잠시 띄우는 명령을 제공한다. 시스템에 `lighttpd`나 `webrick` 같은 경량 웹서버가 설치돼 있어야 이 명령을 사용할 수 있다. 리눅스에서는 `lighttpd`가 설치돼 있을 확률이 높고 프로젝트 디렉토리에서 그냥 `git instaweb`을 실행하면 바로 실행된다. Mac의 Leopard 버전은 Ruby가 미리 설치돼 있기 때문에 `webrick`이 더 낫다. lighttpd이 아니라면 아래와 같이 `--httpd` 옵션을 사용해야 한다:

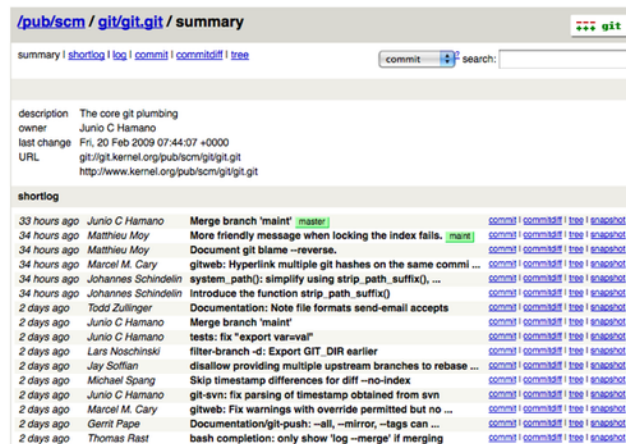



그림 4.1: Git 웹용 UI, GitWeb

```
$ git instaweb --httpd=webrick
[2009-02-21 10:02:21] INFO  WEBrick 1.3.1
[2009-02-21 10:02:21] INFO  ruby 1.8.6 (2008-03-03) [universal-darwin9.0]
```

1234 포트로 HTTPD 서버를 시작하고 이 페이지를 여는 웹브라우저를 자동으로 실행시킨다. 꽤 편리하다. 필요한 일을 모두 마치고 나서 같은 명령어에 `--stop` 옵션을 추가하여 서버를 중지한다:

```
$ git instaweb --httpd=webrick --stop
```

항상 접속가능한 웹 인터페이스를 운영하려면 먼저 웹서버에 이 CGI 스크립트를 설치해야 한다. `apt`나 `yum`으로도 `gitweb`을 설치할 수 있지만, 여기에서는 수동으로 설치한다. 먼저 GitWeb이 포함된 Git 소스 코드를 구한 다음 CGI 스크립트를 빌드한다:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
        prefix=/usr gitweb/gitweb.cgi
$ sudo cp -Rf gitweb /var/www/
```

빌드할 때 `GITWEB_PROJECTROOT` 변수로 Git 저장소의 위치를 알려줘야 한다. 이제 Apache가 이 스크립트를 사용하도록 VirtualHost 항목을 설정한다:

```
<VirtualHost *:80>
    ServerName gitserver
    DocumentRoot /var/www/gitweb
    <Directory /var/www/gitweb>
```

```
        Options ExecCGI +FollowSymLinks +SymLinksIfOwnerMatch
        AllowOverride All
        order allow,deny
        Allow from all
        AddHandler cgi-script cgi
        DirectoryIndex gitweb.cgi
    </Directory>
</VirtualHost>
```

다시 말해서 GitWeb은 CGI를 지원하는 웹서버라면 아무거나 사용할 수 있다. 이제 에 접속하여 온라인으로 저장소를 확인할 수 있을 뿐만 아니라 를 통해서 HTTP 프로토콜로 저장소를 Clone하고 Fetch할 수 있다.

4.7 Gitosis

처음에는 모든 사용자의 공개키를 `authorized_keys`에 저장하는 방법으로도 불편하지 않을 것이다. 하지만, 사용자가 수백 명이 넘으면 관리하기가 매우 고통스럽다. 사용자를 추가할 때마다 매번 서버에 접속할 수도 없고 권한 관리도 안된다. `authorized_keys`에 등록된 모든 사용자는 누구나 프로젝트를 읽고 쓸 수 있다.

이 문제는 매우 널리 사용되고 있는 Gitosis라는 소프트웨어로 해결할 수 있다. Gitosis는 기본적으로 `authorized_keys` 파일을 관리하고 접근제어를 돕는 스크립트 패키지다. 사용자를 추가하고 권한을 관리하는 UI가 웹 인터페이스가 아니라 일종의 Git 저장소라는 점이 재미있다. 프로젝트 설정을 Push하면 그 설정이 Gitosis에 적용된다. 신비롭다!

Gitosis를 설치하기가 쉽지는 않지만 그렇다고 어렵지도 않다. Gitosis는 리눅스에 설치하는 것이 가장 쉽다. 여기서는 Ubuntu 8.10 서버를 사용한다.

Gitosis는 Python이 필요하기 때문에 먼저 Python setuptools 패키지를 설치해야 한다. Ubuntu에서는 아래와 같이 설치한다:

```
$ apt-get install python-setuptools
```

그리고 Gitosis 프로젝트 사이트에서 Gitosis를 Clone한 후 설치한다:

```
$ git clone https://github.com/tv42/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

Gitosis가 설치되면 Gitosis는 저장소 디렉토리로 `/home/git`를 사용하려고 한다. 이대로 사용해도 괜찮지만, 우리의 저장소는 이미 `/opt/git`에 있다. 다시 설정하지 말고 아래와 같이 간단하게 심볼릭 링크를 만들자:

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis가 키들을 관리할 것이기 때문에 현재 파일은 삭제하고 다시 추가해야 한다. 이제부터는 Gitosis가 `authorized_keys`파일을 자동으로 관리할 것이다. `authorized_keys` 파일을 백업해두자:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

그리고 `git` 계정의 쉘을 `git-shell`로 변경했었다면 원래대로 복원해야 한다. Gitosis가 대신 이 일을 맡아줄 것이기 때문에 복원해도 사람들은 여전히 로그인할 수 없다. `/etc/passwd` 파일의 다음 줄을:

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

아래와 같이 변경한다:

```
git:x:1000:1000::/home/git:/bin/sh
```

이제 Gitosis를 초기화할 차례다. `gitosis-init` 명령을 공개키와 함께 실행한다. 만약 공개키가 서버에 없으면 공개키를 서버로 복사해와야 한다:

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

이 명령으로 등록하는 키의 사용자는 Gitosis를 제어하는 파일들이 있는 Gitosis 설정 저장소를 수정할 수 있게 된다. 그리고 수동으로 `post-update` 스크립트에 실행권한을 부여한다:

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

모든 준비가 끝났다. 설정이 잘 됐으면 추가한 공개키의 사용자로 SSH 서버에 접속했을 때 아래와 같은 메시지를 보게 된다:

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
ERROR:gitosis.serve.main:Need SSH_ORIGINAL_COMMAND in environment.
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
  Connection to gitserver closed.
```

이것은 접속을 시도한 사용자가 누구인지 식별할 수는 있지만, Git 명령이 아니어서 거절한다는 뜻이다. 그러니까 실제 Git 명령어를 실행시켜보자. Gitosis 제어 저장소를 Clone한다:

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

`gitosis-admin`이라는 디렉토리가 생긴다. 디렉토리 내용은 크게 두 가지로 나눌 수 있다:

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

`gitoiss.conf` 파일은 사용자, 저장소, 권한 등을 명시하는 설정파일이다. `keydir` 디렉토리에는 저장소에 접근할 수 있는 사용자의 공개키가 저장된다. 사용자마다 공개키가 하나씩 있고 이 공개키로 서버에 접근한다. 이 예제에서는 `scott.pub`이지만 `keydir` 안에 있는 파일의 이름은 사용자마다 다르다. 파일 이름은 `gitosis-init` 스크립트로 공개키를 추가할 때 결정되는데 공개키 끝 부분에 있는 이름이 사용된다.
이제 `gitosis.conf` 파일을 열어보자. 지금 막 Clone한 `gitosis-admin` 프로젝트에 대한 정보만 들어 있다:

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
members = scott
writable = gitosis-admin
```

scott이라는 사용자는 Gitosis를 초기화할 때 사용한 공개키의 사용자이다. 이 사용자만 `gitosis-admin` 프로젝트에 접근할 수 있다.
이제 프로젝트를 새로 추가해보자. `mobile` 단락을 추가하고 그 프로젝트에 속한 개발자나 프로젝트에 접근해야 하는 사용자를 추가한다. 현재는 scott이외에 다른 사용자가 없으니 `scott`만 추가한다. 그리고 `iphone_project` 프로젝트를 새로 추가한다:

```
[group mobile]
writable = iphone_project
members = scott
```

`gitosis-admin` 프로젝트를 수정하면 커밋하고 서버에 Push해야 수정한 설정이 적용된다:

```
$ git commit -am 'add iphone_project and mobile group'
[master 8962da8] add iphone_project and mobile group
 1 file changed, 4 insertions(+)
$ git push origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 272 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:gitosis-admin.git
   fb27aec..8962da8  master -> master
```

로컬에 있는 `iphone_project` 프로젝트에 이 서버를 리모트 저장소로 추가하고 Push하면 서버에 새로운 저장소가 추가된다. 서버에 프로젝트를 새로 만들 때 이제는 수동으로 Bare 저장소를 만들 필요가 없다. 처음 Push할 때 Gitosis가 알아서 생성해 준다:

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
```

Writing objects: 100% (3/3), 230 bytes | 0 bytes/s, done. Total 3 (delta 0), reused 0 (delta 0) To git@gitserver:iphone_project.git * [new branch] master -> master

Gitosis를 이용할 때에는 저장소 경로를 명시할 필요도 없고 사용할 수도 없다. 단지 콜론 뒤에 프로젝트 이름만 적어도 Gitosis가 알아서 찾아 준다.
동료와 이 프로젝트를 공유하려면 동료의 공개키도 모두 추가해야 한다. `~/.ssh/authorized_keys` 파일에 수동으로 추가하는 게 아니라 `keydir` 디렉토리에 하나의 공개키를 하나의 파일로 추가한다. 이 공개키의 파일이름이 `gitosis.conf` 파일에서 사용하는 사용자 이름을 결정한다. John, Josie, Jessica의 공개키를 추가해 보자:

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

이 세 사람을 모두 mobile 팀으로 추가하여 `iphone_project` 에 대한 읽기, 쓰기를 허용한다:

```
[group mobile]
members = scott john josie jessica
writable = iphone_project
```

이 파일을 커밋하고 Push하고 나면 네 명 모두 `iphone_project`를 읽고 쓸 수 있게 된다.
Gitosis의 접근제어 방법은 매우 단순하다. 만약 이 프로젝트에 대해서 John은 읽기만 가능하도록 설정하려면 아래와 같이 한다:

```
[group mobile]
members = scott josie jessica
writable = iphone_project

[group mobile_ro]
members = john
readonly = iphone_project
```

이제 John은 프로젝트를 Clone하거나 Fetch할 수는 있지만, 프로젝트에 Push할 수는 없다. 다양한 사용자와 프로젝트가 있어도 필요한 만큼 그룹을 만들어 사용하면 된다. 그리고 members 항목에 사용자 대신 그룹명을 사용할 수도 있다. 그룹명 앞에 `@`를 붙이면 그 그룹의 사용자를 그대로 상속한다:

```
[group mobile_committers]
members = scott josie jessica

[group mobile]
members   = @mobile_committers
writable  = iphone_project

[group mobile_2]
members   = @mobile_committers john
writable  = another_iphone_project
```

`[gitosis]` 절에 `loglevel=DEBUG`라고 적으면 문제가 생겼을 때 해결하는데 도움이 된다. 그리고 설정이 꼬여버려서 Push할 수 없게 되면 서버에 있는 파일을 수동으로 고쳐도 된다. Gitosis는 `/home/git/.gitosis.conf` 파일의 정보를 읽기 때문에 이 파일을 고친다. `gitosis.conf`는 Push할 때 그 위치로 복사되기 때문에 수동으로 고친 파일은 `gitosis-admin` 프로젝트가 다음에 Push될 때까지 유지된다.

4.8 Gitolite

이 절에서는 Gitolite가 뭐고 기본적으로 어떻게 설치하는 지를 살펴본다. 물론 Gitolite에 들어 있는 Gitolite 문서는 양이 많아서 이 절에서 모두 다룰 수 없다. Gitolite는 계속 진화하고 있기 때문에 이 책의 내용과 다를 수 있다. 최신 내용은 여기에서 확인해야 한다.
Gitolite은 간단히 말해 Git 위에서 운영하는 권한 제어(Authorization) 도구다. 사용자 인증(Authentication)은 `sshd`와 `httpd`를 사용한다. 접속한 접속하는 사용자가 누구인지 가려내는 것이 인증(Authentication)이고 리소스에 대한 접근 권한을 가려내는 일은 권한 제어(Authorization)이다.
Gitolite는 저장소뿐만 아니라 저장소의 브랜치나 태그에도 권한을 명시할 수 있다. 즉, 어떤 사람은 refs(브랜치나 태그)에 Push할 수 있고 어떤 사람은 할 수 없게 하는 것이 가능하다.

4.8.1 설치하기

별도 문서를 읽지 않아도 유닉스 계정만 하나 있으면 Gitolite를 쉽게 설치할 수 있다. 이 글은 여러 가지 리눅스들과 솔라리스 10에서 테스트를 마쳤다. Git, Perl, OpenSSH가 호환되는 SSH 서버가 설치돼 있으면 root 권한도 필요 없다. 앞서 사용했던 `gitserver`라는 서버와 그 서버에 `git` 계정을 만들어 사용한다.

Gitolite는 보통의 서버 소프트웨어와는 달리 SSH를 통해서 접근한다. 서버의 모든 계정은 근본적으로 "Gitolite 호스트"가 될 수 있다. 이 책에서는 가장 간단한 설치 방법으로 설명한다. 자세한 설명 문서는 Gitolite의 문서를 참고한다.

먼저 서버에 `git` 계정을 만들고 `git` 계정으로 로그인 한다. 사용자의 SSH 공개키(`ssh-keygen`으로 생성한 SSH 공개키는 기본적으로 `~/.ssh/id_rsa.pub`에 위치함)를 복사하여 `<이름>.pub` 파일로 저장한다(이 책의 예제에서는 `scott.pub`로 저장). 그리고 다음과 같은 명령을 실행한다:

```
$ git clone git://github.com/sitaramc/gitolite
$ gitolite/install -ln
    # $HOME/bin가 이미 $PATH에 등록돼있다고 가정
$ gitolite setup -pk $HOME/scott.pub
```

마지막 명령은 `gitolite-admin`라는 새 Git 저장소를 서버에 만든다.

다시 작업하던 환경으로 돌아가서 `git clone git@gitserver:gitolite-admin` 명령으로 서버의 저장소를 Clone 했을 때 문제없이 Clone 되면 Gitolite가 정상적으로 설치된 것이다. 이 `gitolite-admin` 저장소의 내용을 수정하고 Push하여 Gitolite을 설치를 마치도록 한다.

4.8.2 자신에게 맞게 설치하기

보통은 기본설정으로 빠르게 설치하는 것으로 충분하지만, 자신에게 맞게 고쳐서 설치할 수 있다. 일부 설정은 주석이 잘 달려있는 rc 파일을 간단히 고쳐서 쓸 수 있지만, 자세한 설정을 위해서는 Gitolite가 제공하는 문서를 살펴보도록 한다.

4.8.3 설정 파일과 접근제어 규칙

설치가 완료되면 홈 디렉토리에 Clone한 `gitolite-admin` 디렉토리로 이동해서 어떤 것들이 있는지 한번 살펴보자:

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/scott.pub
$ cat conf/gitolite.conf

repo gitolite-admin
    RW+                 = scott
```

```
repo testing
    RW+                 = @all
```

gitolite setup 명령을 실행했을 때 주었던 공개키 파일의 이름인 scott은 gitolite-admin 저장소에 대한 읽기와 쓰기 권한을 갖도록 공개키가 등록돼 있다.

사용자를 새로 추가하기도 쉽다. "alice"라는 사용자를 새로 등록하려면 우선 등록할 사람의 공개키 파일을 얻어서 alice.pub라는 이름으로 gitolite-admin 디렉토리 아래에 keydir 디렉토리에 저장한다. 새로 추가한 이 파일을 Add하고 커밋한 후 Push를 하면 alice라는 사용자가 등록된 것이다.

Gitolite의 설정 파일인 conf/example.conf에 대한 내용은 Gitolite 문서에 설명이 잘 되어 있다. 이 책에서는 주요 일부 설정에 대해서만 간단히 살펴본다.

저장소의 사용자 그룹을 쉽게 만들 수 있다. 이 그룹은 매크로와 비슷하다. 그룹을 만들 때는 그 그룹이 프로젝트의 그룹인지 사용자의 그룹인지 구분하지 않지만 사용할 때에는 다르다.

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admins         = scott
@interns        = ashok
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

그리고 ref 단위로 권한을 제어한다. 다음 예제를 보자. 인턴(interns)은 int 브랜치만 Push할 수 있고 engineers는 eng-로 시작하는 브랜치와 rc 뒤에 숫자가 붙는 태그만 Push할 수 있다. 그리고 관리자는 모든 ref에 무엇이든지(되돌리기도 포함됨) 할 수 있다.

```
repo @oss_repos
    RW  int$                = @interns
    RW  eng-                = @engineers
    RW  refs/tags/rc[0-9]   = @engineers
    RW+                     = @admins
```

RW나 RW+ 뒤에 나오는 표현식은 정규표현식(regex)이고 Push하는 ref 이름의 패턴을 의미한다. 그래서 우리는 refex라고 부른다. 물론 refex는 여기에 보여준 것보다 훨씬 더 강력하다. 하지만, 펄의 정규표현식에 익숙하지 않은 독자도 있으니 여기서는 무리하지 않았다.

그리고 이미 예상했겠지만 Gitolite는 refs/heads/라고 시작하지 않는 refex에 대해서는 암묵적으로 refs/heads/가 생략된 것으로 판단한다.

특정 저장소에 사용하는 규칙을 한 곳에 모아 놓지 않아도 괜찮다. 위에 보여준 oss_repos 저장소 설정과 다른 저장소 설정이 마구 섞여 있어도 괜찮다. 아래와 같이 목적이 분명하고 제한적인 규칙을 아무 데나 추가해도 좋다:

```
repo gitolite
    RW+                     = sitaram
```

이 규칙은 `gitolite` 저장소를 위해 지금 막 추가한 규칙이다.
이제는 접근제어 규칙이 실제로 어떻게 적용되는지 설명한다. 이제부터 그 내용을 살펴보자.
Gitolite는 접근 제어를 두 단계로 한다. 첫 단계가 저장소 단계인데 접근하는 저장소의 ref 중에서 하나라도 읽고 쓸 수 있으면 실제로 그 저장소 전부에 대해 읽기, 쓰기 권한이 있는 것이다.
두 번째 단계는 브랜치나 태그 단위로 제어하는 것으로 오직 "쓰기" 접근만 제어할 수 있다. 어느 사용자가 특정 ref 이름으로 접근을 시도하면(`W`나 `+`같은) 설정 파일에 정의된 순서대로 접근 제어 규칙이 적용된다. 그 순서대로 사용자 이름과 ref 이름을 비교하는데 ref 이름의 경우 단순히 문자열을 비교하는 것이 아니라 정규 표현식으로 비교한다. 해당되는 것을 찾으면 정상적으로 Push되지만 찾지 못하면 거절된다.

4.8.4 "deny" 규칙을 꼼꼼하게 제어하기

지금까지는 `R`, `RW`, `RW+` 권한에 대해서만 다뤘다. Gitolite는 "deny" 규칙을 위해서 `-` 권한도 지원한다. 이것으로 복잡도를 낮출 수 있다. `-`로 거절도 할 수 있기 때문에 규칙의 순서가 중요하다.
다시 말해서 engineers가 master와 integ 브랜치 이외의 모든 브랜치를 되돌릴 수 있게 하고 싶으면 아래와 같이 한다:

```
    RW  master integ    = @engineers
    -   master integ    = @engineers
    RW+                 = @engineers
```

즉, 접근제어 규칙을 순서대로 찾기 때문에 순서대로 정의해야 한다. 첫 번째 규칙은 master나 integ 브랜치에 대해서 읽기, 쓰기만 허용하고 되돌리기는 허용하지 않는다. master나 integ 브랜치를 되돌리는 Push는 첫 번째 규칙에 어긋나기 때문에 바로 두 번째 규칙으로 넘어간다. 그리고 거기서 거절된다. master나 integ 브랜치 이외 다른 ref에 대한 Push는 첫 번째 규칙과 두 번째 규칙에는 만족하지 않고 마지막 규칙으로 허용된다.

4.8.5 파일 단위로 Push를 제어하기

브랜치 단위로 Push를 제어할 수 있지만 수정된 파일 단위로도 제어할 수 있다. 예를 들어 Makefile을 보자. Makefile 파일에 의존하는 파일은 매우 많고 보통 꼼꼼하게 수정하지 않으면 문제가 생긴다. 그래서 아무나 Makefile을 수정하게 둘 수 없다. 그러면 아래와 같이 설정한다:

```
repo foo
    RW                      =   @junior_devs @senior_devs

    -   VREF/NAME/Makefile  =   @junior_devs
```

예전 버전의 Gitolite에서 버전을 올리려는 사용자는 설정이 많이 달라진 것을 알게 될 것이다. Gitolite의 버전업 가이드를 필히 참고하자.

4.8.6 Personal 브랜치

Gitolite는 또 “Personal 브랜치”라고 부르는 기능을 지원한다. 이 기능은 실제로 “Personal 브랜치 네임스페이스”라고 부르는 것이 더 적절하다. 이 기능은 기업에서 매우 유용하다.

Git을 사용하다 보면 코드를 공유하려고 “Pull 해주세요”라고 말해야 하는 일이 자주 생긴다. 그런데 기업에서는 인증하지 않은 접근을 절대 허용하지도 않는 데다가 아예 다른 사람의 컴퓨터에 접근할 수 없다. 그래서 공유하려면 중앙 서버에 Push하고 나서 Pull해야 한다고 다른 사람에게 말해야만 한다.

중앙집중식 VCS에서 이렇게 마구 사용하면 브랜치 이름이 충돌할 확률이 높다. 그때마다 관리자는 추가로 권한을 관리해줘야 하기 때문에 관리자의 노력이 쓸데없이 낭비된다.

Gitolite는 모든 개발자가 “personal”이나 “scratch” 네임스페이스를 가질 수 있도록 허용한다. 이 네임스페이스는 `refs/personal/<devname>/*` 라고 표현한다. 자세한 내용은 Gitolite 문서를 참고한다.

4.8.7 “와일드카드” 저장소

Gitolite는 펄 정규표현식으로 저장소 이름을 표현하기 때문에 와일드카드를 사용할 수 있다. 그래서 `assignments/s[0-9][0-9]/a[0-9][0-9]` 같은 정규표현식을 사용할 수 있다. 사용자가 새로운 저장소를 만들 수 있는 새로운 권한 모드인 `C` 모드를 사용할 수도 있다. 저장소를 새로 만든 사람에게는 자동으로 접근 권한이 부여된다. 다른 사용자에게 접근 권한을 주려면 `R` 또는 `RW` 권한을 설정한다. 다시 한번 말하지만 문서에 모든 내용이 다 있으므로 꼭 보기를 바란다.

4.8.8 그 밖의 기능들

마지막으로 알고 있으면 유용한 것들이 있다. Gitolite에는 많은 기능이 있고 자세한 내용은 “Faq, Tip, 등등”의 다른 문서에 잘 설명돼 있다.

로깅: 누군가 성공적으로 접근하면 Gitolite는 무조건 로그를 남긴다. 관리자가 한눈파는 사이에 되돌리기(`RW+`) 권한을 가진 망나니가 `master` 브랜치를 날려버릴 수 있다. 이 경우 로그 파일이 구원해준다. 이 로그 파일을 참고하여 버려진 SHA를 빠르고 쉽게 찾을 수 있다.

접근 권한 보여주기: 만약 어떤 서버에서 작업을 시작하려고 할 때 필요한 것이 무엇일까? Gitolite는 해당 서버에 대해 접근할 수 있는 저장소가 무엇인지, 어떤 권한을 가졌는지 보여준다:

```
hello scott, this is git@git running gitolite3 v3.01-18-g9609868 on git 1.7.4.4

     R     anu-wsd
     R     entrans
     R  W  git-notes
     R  W  gitolite
     R  W  gitolite-admin
     R     indic_web_input
     R     shreelipi_converter
```

권한 위임: 조직 규모가 크면 저장소에 대한 책임을 여러 사람이 나눠 가지는 게 좋다. 여러 사람이 각자 맡은 바를 관리하도록 한다. 그래서 주요 관리자의 업무가 줄어들기에 병목현상이 적어진다. 이 기

능에 대해서는 `doc/` 디렉토리에 포함된 Gitolite 문서를 참고하라.
미러링: Gitolite의 미러는 여러 개 만들 수 있어서 주 서버가 다운 돼도 변경하면 된다.

4.9 Git 데몬

공개된 프로젝트는 누가 읽기 접근을 시도하는지 알 필요가 없다. 그래서 HTTP 프로토콜을 사용하거나 Git 프로토콜을 사용해야 한다. Git 프로토콜이 HTTP 프로토콜보다 효율적이기 때문에 속도가 빠르다. 따라서 사용자의 시간을 절약해 준다.
다시 강조하지만, 이것은 불특정 다수에게 읽기 접근을 허용할 때에만 쓸만하다. 만약 서버가 외부에 그냥 노출돼 있으면 우선 방화벽으로 보호하고 프로젝트만 외부에서 접근할 수 있게 만든다. 서버를 방화벽으로 보호하고 있으면 CI 서버나 빌드 서버같은 컴퓨터나 사람이 읽기 접근이 가능하도록 설정한다. 모두 SSH 키를 일일이 추가하고 싶지 않을 때 이 방법을 사용한다.
어쨌든 Git 프로토콜은 상대적으로 설치하기 쉽다. 그냥 데몬을 실행한다:

```
git daemon --reuseaddr --base-path=/opt/git/ /opt/git/
```

`--reuseaddr`는 서버가 기존의 연결이 타임아웃될 때까지 기다리지 말고 바로 재시작하게 하는 옵션이다. `--base-path` 옵션을 사용하면 사람들이 프로젝트를 Clone할 때 전체 경로를 사용하지 않아도 된다. 그리고 마지막에 있는 경로는 노출할 저장소의 위치다. 마지막으로 방화벽을 사용하고 있으면 9418 포트를 열어서 지금 작업하는 서버의 숨통을 틔워준다.
운영체제에 따라 Git 데몬을 실행시키는 방법은 다르다. 우분투에서는 Upstart 스크립트를 사용한다. 아래와 같이 파일을 만들고:

```
/etc/event.d/local-git-daemon
```

다음의 내용을 입력한다:

```
start on startup
stop on shutdown
exec /usr/bin/git daemon \
    --user=git --group=git \
    --reuseaddr \
    --base-path=/opt/git/ \
    /opt/git/
respawn
```

저장소를 읽을 수만 있는 사용자로 데몬을 실행시킬 것을 보안을 위해 강력하게 권고한다. `git-ro`라는 계정을 새로 만들고 그 계정으로 데몬을 실행시킨다. 여기에서는 쉽게 설명하려고 그냥 Gitosis를 실행했던 `git` 계정으로 실행시킨다.
서버가 재시작할 때 Git 데몬이 자동으로 실행되고 데몬이 죽어도 자동으로 재시작된다. 서버는 놔두고 Git 데몬만 재시작할 수 있다:

```
initctl start local-git-daemon
```

다른 시스템에서는 `sysvinit` 시스템의 `xinetd` 스크립트를 사용하거나 자신만의 방법으로 해야 한다.

Git 데몬을 통해서 아무나 읽을 수 있다는 것을 Gitosis 서버에 알려주어야 한다. Git 데몬으로 읽기 접근을 허용하는 저장소가 무엇인지 설정에 추가해야 한다. 만약 `iphone_project`에 Git 프로토콜을 허용했다면 아래와 같은 것을 `gitosis.conf` 파일의 하단에 추가한다:

```
[repo iphone_project]
daemon = yes
```

차례대로 커밋과 Push하고 나면 지금 실행 중인 데몬이 9418 포트로 접근하는 사람에게 서비스하기 시작한다.

Gitosis 없이도 Git 데몬을 설치할 수 있지만 그러려면 서비스하고자 하는 프로젝트마다 아래와 같이 `git-daemon-export-ok` 파일을 넣어 주어야 한다:

```
$ cd /path/to/project.git
$ touch git-daemon-export-ok
```

이 파일이 있으면 Git 데몬은 인증 없이 프로젝트를 노출하는 것으로 판단한다.

또한, Gitweb으로 노출하는 프로젝트도 Gitosis로 제어할 수 있다. 먼저 `/etc/gitweb.conf` 파일에 아래와 같은 내용을 추가해야 한다:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Gitosis 설정 파일에 `gitweb` 설정을 넣거나 빼면 사용자는 GitWeb을 통해 프로젝트를 볼 수도 있고 못 볼 수도 있다.

```
[repo iphone_project]
daemon = yes
gitweb = yes
```

이제 이것을 커밋하고 Push하면 GitWeb을 통해 `iphone_project`를 볼 수 있다.

4.10 Hosted Git

Git 서버를 설치하는 등의 일을 하고 싶지 않으면 전문 호스팅 사이트를 이용하면 된다. 호스팅 사이트는 몇 가지 장점이 있다. 설정이 쉬워서 바로 프로젝트를 시작할 수 있을 뿐만 아니라 직접 서버를 관리하고 모니터링하지 않아도 된다. 내부적으로 Git 서버를 직접 설치하고 운영하고 있어도 오픈소스 프로젝트는 호스팅 사이트를 이용하는 것이 좋다. 이렇게 하면 보통 오픈소스 커뮤니티로부터 좀 더 쉽게 도움받을 수 있다.

요즘은 이용할 수 있는 호스팅 사이트들이 많다. 각각 장단점이 있기 때문에 다음 페이지에서 최신 정보를 확인해보자:

```
https://git.wiki.kernel.org/index.php/GitHosting
```

이 절에서 전부 설명할 수는 없고(필자는 저 회사 중 한군데에서 일한다) GitHub에 계정과 프로젝트를 만드는 방법을 설명한다.

GitHub은 가장 큰 오픈소스 Git 호스팅 사이트이고 공개(Public) 프로젝트와 비공개(Private) 프로젝트에 대한 호스팅 서비스를 제공하는 보기 드문 사이트다. 그래서 상업용 비공개 코드와 공개 코드를 같은 곳에 둘 수 있다. 실제로 이 책도 GitHub에서 비공개로 작성했다.

4.10.1 GitHub

GitHub는 프로젝트 네임스페이스가 다른 코드 호스팅 사이트들과 다르다. GitHub는 프로젝트가 아니라 사용자가 중심이다. GitHub에 `grit` 프로젝트를 호스팅하고 싶으면 `github.com/grit`이 아니라 `github.com/schacon/grit`으로 접속해야 한다. 그리고 처음 프로젝트를 시작한 사람이 그 프로젝트를 잊어버려도 누구나 프로젝트를 이어갈 수 있다. 프로젝트 주 저장소라고 해서 특별한게 아니다.

GitHub은 이윤을 목적으로 하는 회사이기 때문에 비공개 저장소를 만들려면 돈을 내야 한다. 하지만, 누구나 손쉽게 무료 계정을 만들어 오픈소스 프로젝트를 시작할 수 있다. 어떻게 사용하는지 간략하게 설명한다.

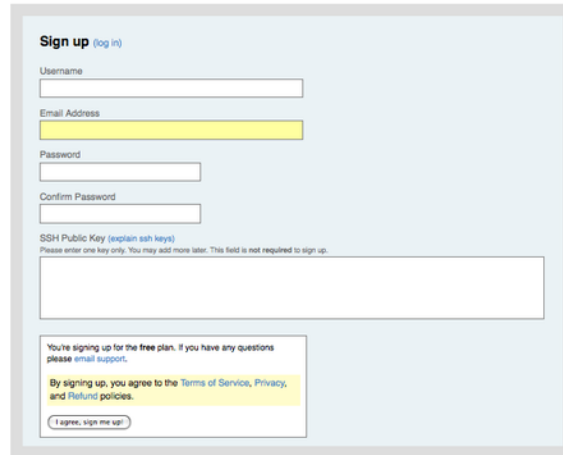
4.10.2 계정 설정하기

먼저 무료 계정을 하나 만든다. 가격 정책에 대해 알려주며 가입을 시작할 수 있는 `https://github.com/pricing`에 방문하여 “Sign up” 버튼을 클릭한다. 그러면 가입 페이지로 이동한다.

그림 4.2: GitHub 가격 정책 페이지

아직 등록되지 않은 사용자 이름을 입력하고 e-mail 주소와 암호를 입력한다(그림 4-3).

그리고 SSH 공개키가 있으면 바로 등록한다. SSH 키를 만드는 방법은 “바로 설정하기” 절에서 이미 설명했다. 그 공개키 파일의 내용을 복사해서 SSH 공개키 입력 박스에 붙여 넣는다. “explain ssh

그림 4.3: GitHub 가입 폼

keys” 링크를 클릭하면 key를 생성하는 방법을 자세히 설명해준다. 주요 운영체제에서 하는 방법이 모두 설명돼 있다. “I agree, sign me up” 버튼을 클릭하면 자신만의 대쉬보드 페이지가 나타난다.

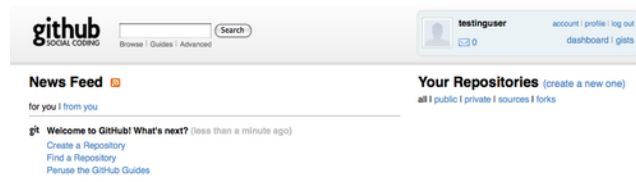
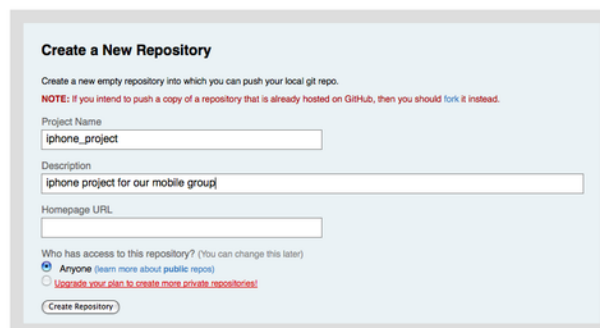



그림 4.4: GitHub 사용자 대쉬보드

그리고 저장소를 만들자.

4.10.3 저장소 만들기

`Your Repositories`옆에 있는 “create a new one” 링크를 클릭하면 저장소를 만드는 입력 폼을 볼 수 있다(그림 4-5).

그림 4.5: GitHub의 저장소를 생성하는 폼

이 폼에 프로젝트 이름과 프로젝트 설명을 적는다. 다 적은 후에 “Create Repository” 버튼을 클릭하면 GitHub에 저장소가 생긴다.

이 저장소에는 아직 코드가 없어서 GitHub은 프로젝트를 새로 만드는 방법, 이미 있는 Git 프로젝트를 Push하는 법, 공개된 Subversion 저장소에서 프로젝트를 가져오는(Import) 방법 등을 보여준다.

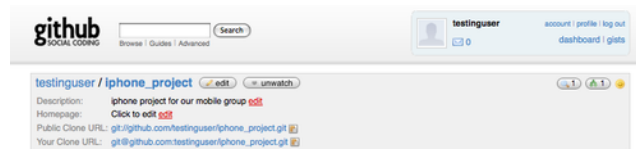



그림 4.6: GitHub 프로젝트 정보

그림 4.7: 새 저장소를 위한 사용설명서

여기 설명하는 내용은 이미 우리가 배웠다. 프로젝트가 없을 때는 아래와 같이 프로젝트를 초기화한다:

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

만약 이미 로컬에 Git 저장소가 있으면 GitHub 저장소를 리모트 저장소로 등록하고 master 브랜치를 Push한다:

```
$ git remote add origin git@GitHub.com:testinguser/iphone_project.git
$ git push origin master
```

이제 프로젝트가 GitHub에서 서비스되니 공유하고 싶은 사람에게 URL을 알려 주면 된다. URL은 이다. 그리고 이 저장소의 정보를 잘 살펴보면 Git URL이 두 개라는 것을 발견할 수 있다.

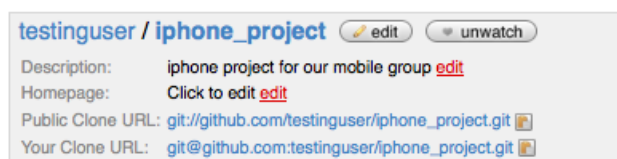



그림 4.8: 프로젝트의 공개 URL과 비공개 URL

`Public Clone URL`은 말 그대로 누구나 프로젝트를 Clone할 수 있도록 모두에게 읽기 전용으로 공개하

는 것이다. 이 URL을 다른 사람에 알려주거나 웹사이트 같은데 공개하는 것을 부담스러워 하지 않아도 된다.
`Your Clone URL`은 읽고 쓸 수 있는 SSH 기반 URL이다. 사용자 계정에 등록한 공개키와 한 짝인 개인키로만 접속할 수 있다. 다른 사용자로 이 프로젝트에 방문하면 이 URL은 볼 수 없고 공개 URL만 볼 수 있다.

4.10.4 Subversion으로부터 코드 가져오기(Import)

GitHub은 공개 중인 Subversion 프로젝트를 Git 프로젝트로 만들어 준다. 사용설명서 하단에 있는 "Subversion에서 Import하기" 링크를 클릭하면 임포트 폼을 볼 수 있고 거기에 Subversion 프로젝트의 URL을 넣는다(그림 4-9).

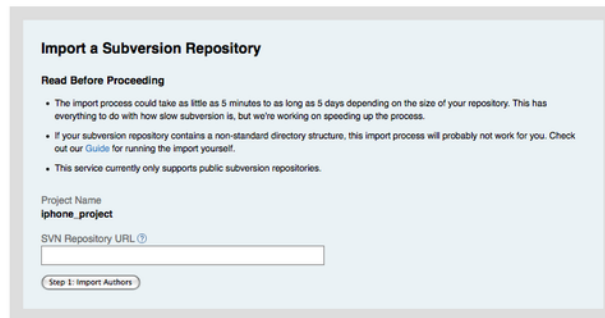



그림 4.9: Subversion 프로젝트를 Import하는 화면

프로젝트가 비표준 방식을 사용하거나 규모가 너무 크고 비공개라면 이 기능을 사용할 수 없다. 7장에서 수동으로 임포트하는 방법에 대해 좀 더 자세히 배운다.

4.10.5 동료 추가하기

동료를 추가하자. 먼저 John씨, Josie씨, Jessica씨를 모두 GitHub에 가입시키고 나서 그들을 동료로 추가하고 저장소에 Push할 수 있는 권한을 준다.
프로젝트 페이지에 있는 Admin 버튼을 클릭해서 관리 페이지로 이동한다(그림 4-10).

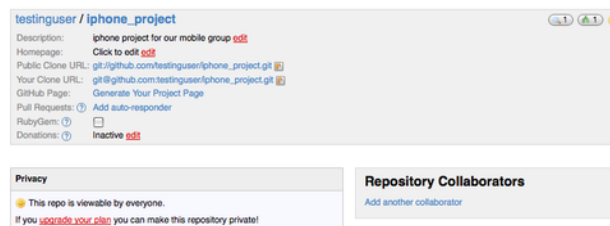



그림 4.10: GitHub의 프로젝트 관리 페이지

다른 사람에게 쓰기 권한을 주려면 "Add another collaborator" 링크를 클릭한다. 그러면 텍스트 박스가 새로 나타나는 데 거기에 사용자 이름을 입력한다. 사용자 이름을 입력하기 시작하면 자동으로 시스템에 존재하는 사용자를 찾아서 보여 준다. 원하는 사용자를 찾으면 Add 버튼을 클릭해서 그 사용자를 동료로 만든다.
추가한 사람은 동료 목록 박스에서 모두 확인할 수 있다(그림 4-12).
그리고 만약 다시 혼자 작업하고 싶어지면 "revoke" 링크를 클릭하여 쫓아낸다. 쫓겨나면 더는 Push할 수 없다. 또 기존 프로젝트에 등록된 동료를 그룹으로 묶어 추가할 수도 있다.

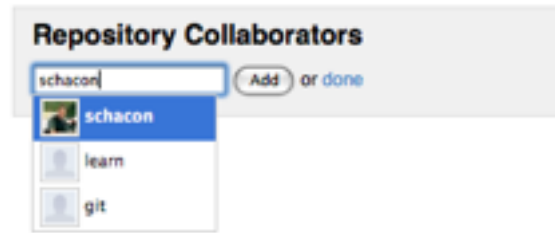



그림 4.11: 프로젝트에 동료 추가하기

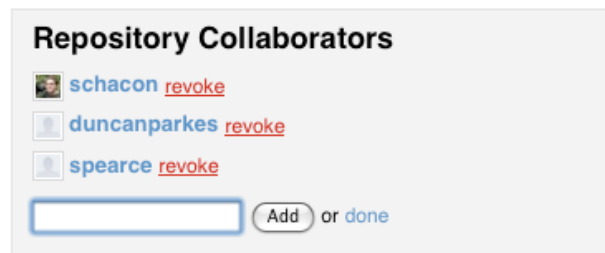



그림 4.12: 프로젝트 동료들

4.10.6 내 프로젝트

Subversion에서 Import했거나 로컬의 프로젝트를 Push하고 나면 프로젝트 메인 페이지가 그림 4-13같이 바뀐다.

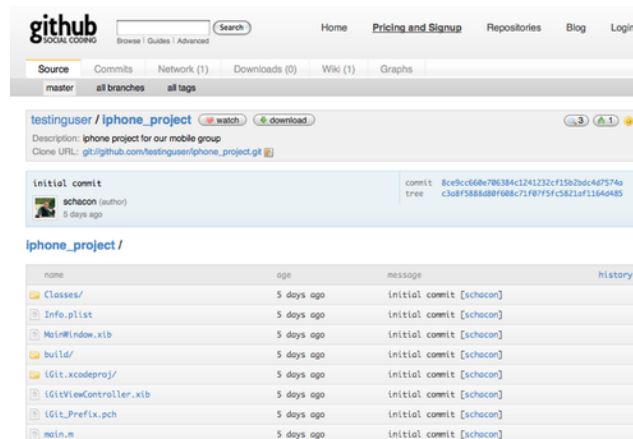



그림 4.13: GitHub의 프로젝트 메인 페이지

사람들이 이 프로젝트에 방문하면 이 페이지가 제일 처음 보인다. 이 페이지는 몇 가지 탭으로 구성된다. Commits 탭은 지금까지의 커밋을 `git log` 명령을 실행시킨 것처럼 최신 것부터 보여준다. Network 탭은 프로젝트를 복제한 사람들과 기여한 사람들을 모두 보여준다. Downloads 탭에는 바이너리 파일이나 프로젝트의 태그 버전을 압축해서 올릴 수 있다. Wiki 탭은 프로젝트에 대한 정보나 문서를 쓰는 곳이다. Graphs 탭은 사람들의 활동을 그림과 통계로 보여준다. 메인 탭인 Source 탭은 프로젝트의 메인 디렉토리를 보여주고 README 파일이 있으면 자동으로 화면에 출력해 준다. 그리고 마지막 커밋 내용도 함께 보여준다.

4.10.7 프로젝트 Fork

권한이 없는 프로젝트에 참여하고 싶으면 GitHub는 프로젝트를 Fork하도록 권고한다. 마침 매우 흥미롭게 보이는 프로젝트를 발견했다고 하자. 그 프로젝트를 조금 뜯어고치려면 프로젝트 페이지 상단에 있는 “fork” 버튼을 클릭한다. 그러면 GitHub는 접속한 사용자의 계정으로 프로젝트를 Fork해 준다. 사용자는 이 프로젝트에 마음대로 Push할 수 있다.

굳이 Push할 수 있도록 사람들을 동료로 추가하지 않아도 된다. 사람들은 마음껏 프로젝트를 Fork하고 Push할 수 있다. 그리고 원래 프로젝트의 관리자는 다른 사람의 프로젝트를 리모트 저장소로 추가하고 그 작업물을 가져와서 Merge한다.

프로젝트 페이지에 들어가서 상단의 “fork” 버튼을 클릭하여 프로젝트를 복제한다(그림 4-14) 그림 4-14의 예는 mojombo/chronic 프로젝트 페이지이다

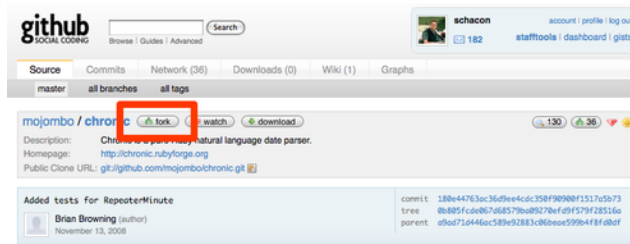



그림 4.14: 어떤 저장소든지 “fork” 버튼을 클릭하면 Push할 수 있는 저장소를 얻을 수 있다

클릭하는 순간, 이 프로젝트를 즉시 Fork한다(그림 4-15).

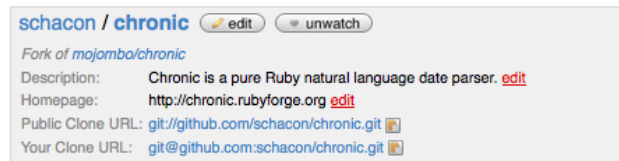



그림 4.15: Fork한 프로젝트

4.10.8 GitHub 요약

빨리 한번 전체를 훑어보는 것이 중요하기 때문에 여기에서는 GitHub에 대해 이 정도로만 설명했다. 몇 분 만에 계정과 프로젝트를 만들고 Push까지 할 수 있다. GitHub에 있는 개발자 커뮤니티 규모는 매우 크기 때문에 만약 GitHub에 오픈 소스 프로젝트를 만들면 다른 개발자들이 당신의 프로젝트를 복제하고 당신을 도울 것이다. GitHub는 Git을 빨리 사용해 볼 수 있도록 돕는다.

4.11 요약

리모트 저장소를 만들고 다른 사람과 협업하거나 작업물을 공개하는 방법은 여러 가지다.

서버를 직접 구축하는 것은 할 일이 많은데다가 방화벽도 필요하다. 그리고 이렇게 서버를 만들고 관리하는 일은 시간이 많이 든다. 호스팅 사이트를 이용하면 쉽게 시작할 수 있다. 하지만, 코드를 타인의 서버에 보관해야 하기 때문에 사용하지 않는 조직들이 많다.

자신의 조직에서 어떤 방법으로 협업해야 할지 고민해야 하는 시점이 되었다.

5장

분산 환경에서의 Git

앞 장에서 다른 개발자와 코드를 공유하는 리모트 저장소를 만드는 법을 배웠다. 로컬에서 작업하는 데 필요한 기본적인 명령어에는 어느 정도 익숙해졌다. 이제는 분산 환경에서 Git이 제공하는 기능을 어떻게 효율적으로 사용할지를 배운다.

이번 장에서는 분산 환경에서 Git을 어떻게 사용할 수 있을지 살펴본다. 프로젝트 기여자 입장과 여러 수정사항을 취합하는 관리자 입장에서 두루 살펴본다. 즉, 프로젝트 기여자 또는 관리자로서 작업물을 프로젝트에 어떻게 포함시킬지와 수 많은 개발자가 수행한 일을 취합하고 프로젝트를 운영하는 방법을 배운다.

5.1 분산 환경에서의 Workflow

중앙집중형 버전 관리 시스템과는 달리 Git은 분산형이다. Git의 구조가 훨씬 더 유연하기 때문에 여러 개발자가 함께 작업하는 방식을 더 다양하게 구성할 수 있다. 중앙집중형 버전 관리 시스템에서 각 개발자는 중앙 저장소를 중심으로 하는 하나의 노드일 뿐이다. 하지만, Git에서는 각 개발자의 저장소가 하나의 노드이기도 하고 중앙 저장소 같은 역할도 할 수 있다. 즉, 모든 개발자는 다른 개발자의 저장소에 일한 내용을 전송하거나, 다른 개발자들이 참여할 수 있도록 자신이 운영하는 저장소 위치를 공개할 수도 있다. 이런 특징은 프로젝트나 팀이 코드를 운영할 때 다양한 Workflow을 만들 수 있도록 해준다. 이런 유연성을 살려 저장소를 운영하는 몇 가지 방식을 소개한다. 각 방식의 장단점을 살펴보고 그 방식 중 하나를 고르거나 여러 가지를 적절히 섞어 쓰면 된다.

5.1.1 중앙집중식 Workflow

중앙집중식 시스템에서는 보통 중앙집중식 협업 모델이라는 한 가지 방식밖에 없다. 중앙 저장소는 딱 하나 있고 변경 사항은 모두 이 중앙 저장소에 집중된다. 개발자는 이 중앙 저장소를 중심으로 작업한다(그림 5-1).

중앙집중식에서 개발자 두 명이 중앙저장소를 Clone하고 각자 수정하는 상황을 생각해보자. 한 개발자가 자신이 한 일을 커밋하고 나서 아무 문제 없이 서버에 Push한다. 그러면 다른 개발자는 자신의 일을 커밋하고 Push하기 전에 첫 번째 개발자가 한 일을 먼저 Merge해야 한다. Merge를 해야 첫 번째 개발자가 작업한 내용을 덮어쓰지 않는다. 이런 개념은 Subversion과 같은 중앙집중식 버전 관리 시스템에서 사용하는 방식이고 Git에서도 당연히 이런 Workflow를 사용할 수 있다.

팀이 작거나 이미 중앙집중식에 적응한 상황이라면 이 Workflow에 따라 Git을 도입하여 사용할 수 있다. 중앙 저장소를 하나 만들고 개발자 모두에게 Push 권한을 부여한다. 모두에게 Push 권한을 부여해도 Git은 한 개발자가 다른 개발자의 작업 내용을 덮어쓰도록 허용하지 않는다. 한 개발자가 Clone하고 나서 수정하는 사이에 이미 다른 개발자가 중앙 저장소에 무언가 Push했다면 이 개발자는 Push할 수

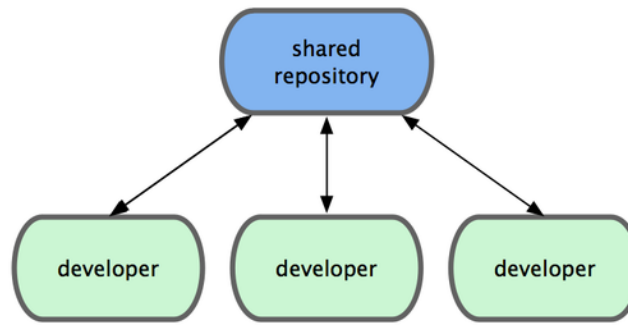



그림 5.1: 중앙집중식 Workflow

없다. Git은 개발자에게 지금 Push하려는 커밋으로 Fast-forward할 수 없으니 Fetch하고 Merge해야 서버로 Push할 수 있다고 알려준다. 이런 개념은 개발자에게 익숙해서 거부감 없이 도입할 수 있다.

5.1.2 Integration-Manager Workflow

Git을 사용하면 리모트 저장소를 여러 개 운영할 수 있다. 다른 개발자는 읽기만 가능하고 자신은 쓰기도 가능한 공개 저장소를 만드는 Workflow도 가능하다. 이 Worlflow에는 보통 프로젝트를 대표하는 하나의 공식 저장소가 있다. 기여자는 우선 공식 저장소를 하나 Clone하고 수정하고 나서 자신의 저장소에 Push한다. 그다음에 프로젝트 Integration-Manager에게 새 저장소에서 Pull하라고 요청한다. 그러면 그 Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고, 로컬에서 기여물을 테스트하고, 프로젝트 메인 브랜치에 Merge하고, 그 내용을 다시 프로젝트 메인 저장소에 Push한다. 이런 과정은 아래와 같다(그림 5-2).

1. 프로젝트 Integration-Manager는 프로젝트 메인 저장소에 Push를 한다.
2. 프로젝트 기여자는 메인 저장소를 Clone하고 수정한다.
3. 기여자는 자신의 저장소에 Push하고 Integration-Manager가 접근할 수 있도록 공개해 놓는다.
4. 기여자는 Integration-Manager에게 변경사항을 적용해 줄 것을 E-mail 같은 것으로 요청한다.
5. Integration-Manager는 기여자의 저장소를 리모트 저장소로 등록하고 수정사항을 Merge하여 테스트한다.
6. Integration-Manager는 Merge한 사항을 메인 저장소에 Push한다.

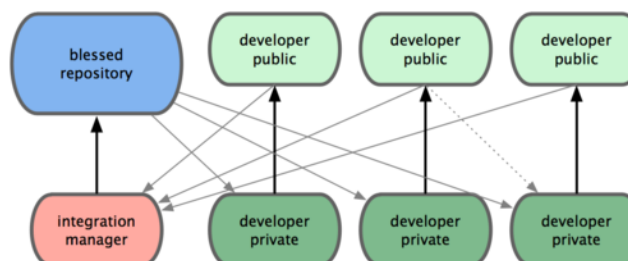



그림 5.2: Integration-Manager Workflow

이 방식은 GitHub 같은 사이트에서 주로 사용하는 방식이다. GitHub는 프로젝트를 Fork하고 수정사항을 반영하여 다시 모두에게 공개하기 좋은 구조로 되어 있다. 이 방식의 장점은 기여자와 Integration-Manager가 각자의 사정에 맞춰 프로젝트를 유지할 수 있다는 점이다. 기여자는 자신의

저장소와 브랜치에서 수정 작업을 계속해 나갈 수 있고 수정사항이 프로젝트에 반영되도록 기다릴 필요가 없다. 관리자는 여유를 가지고 기여자가 Push해 놓은 커밋을 적절한 시점에 Merge한다.

5.1.3 Dictator and Lieutenants Workflow

이 방식은 저장소를 여러개 운영하는 방식을 변형한 구조이다. 보통 수백 명의 개발자가 참여하는 아주 큰 프로젝트를 운영할 때 이 방식을 사용한다. 리눅스 커널 프로젝트가 대표적이다. 여러 명의 Integration-Manager가 저장소에서 자신이 맡은 부분만을 담당하는데 이들을 Lieutenants라고 부른다. 모든 Lieutenant는 최종 관리자 아래에 있으며 이 최종 관리자를 Dictator라고 부른다(그림 5-3).

1. 개발자는 코드를 수정하고 master 브랜치를 기준으로 자신의 토픽 브랜치를 Rebase한다. 여기서 master 브랜치란 Dictator의 브랜치를 말한다.
2. Lieutenant들은 개발자들의 수정사항을 자신이 관리하는 master 브랜치에 Merge한다.
3. Dictator는 Lieutenant의 master 브랜치를 자신의 master 브랜치로 Merge한다.
4. Dictator는 Merge한 자신의 master 브랜치를 Push하여 다른 모든 개발자가 Rebase할 수 있는 기준으로 만든다.

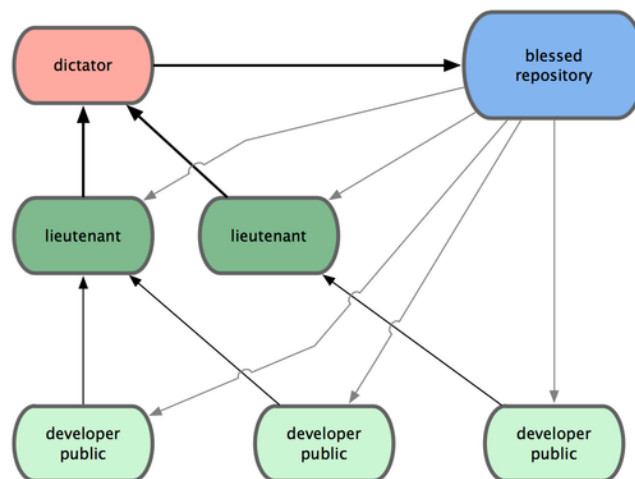



그림 5.3: Benevolent dictator Workflow

이 방식이 일반적이지 않지만 깊은 계층 구조를 가지는 환경이나 규모가 큰 프로젝트에서는 매우 쓸모 있다. 프로젝트 리더가 모든 코드를 통합하기 전에 코드를 부분부분 통합하도록 여러 명의 Lieutenant에게 위임한다.

이 세 가지 Workflow가 Git 같은 분산 버전 관리 시스템에서 주로 사용하는 것들이다. 사실 이런 Workflow뿐만 아니라 다양한 변종 Workflow가 실재로 사용된다. 어떤 방식을 선택하고 혹은 조합해야 하는 지 살짝 감이 잡힐 것이다. 이 장에서는 몇 가지 구체적 사례를 들고 우리가 다양한 환경에서 각 역할을 어떻게 수행하는지 살펴본다.

5.2 프로젝트에 기여하기

이미 다른 장에서 기본적인 Git 사용법에 대해서 배웠고 몇 가지 Workflow도 살펴보았다. 이제 이 절에서는 Git으로 프로젝트에 기여하는 방법에 대해 배운다.

프로젝트에 기여하는 방식이 매우 다양하다는 점을 설명하기란 정말 어렵다. Git이 워낙 유연하게 설계됐기 때문에 사람들은 여러 가지 방식으로 사용할 수 있다. 게다가 프로젝트마다 환경이 달라서 프로

젝트에 기여하는 방식을 쉽게 설명하기란 정말 어렵다. 기여하는 방식에 영향을 끼치는 몇 가지 변수가 있다. 활발히 기여하는 개발자의 수가 얼마인지, 선택한 Workflow가 무엇인지, 각 개발자에게 접근 권한을 어떻게 부여했는지, 외부에서도 기여할 수 있는지 등이 변수다.

첫 번째로 살펴볼 변수는 활발히 활동하는 개발자의 수이다. 얼마나 많은 개발자가 얼마나 자주 코드를 쏟아 내는가 하는 점이 활발한 개발자의 기준이다. 대부분 둘, 셋 정도의 개발자가 하루에 몇 번 커밋을 하고 활발하지 않은 프로젝트는 더 띄엄띄엄할 것이다. 하지만, 아주 큰 프로젝트는 수백, 수천 명의 개발자가 하루에도 수십, 수백 개의 커밋을 만들어 낸다. 개발자가 많으면 많을수록 코드를 깔끔하게 적용하거나 Merge하기 어려워진다. 어떤 것은 다른 개발자가 기여한 것으로 불필요해지기도 하고 때론 서로 충돌이 일어난다. 어떻게 해야 코드를 최신으로 유지하면서 원하는 대로 수정할 수 있을까?

두 번째 변수는 프로젝트에서 선택한 저장소 운영 방식이다. 메인 저장소에 개발자 모두가 쓰기 권한을 가지는 중앙집중형 방식인가? 프로젝트에 모든 Patch를 검사하고 통합하는 관리자가 따로 있는가? 모든 수정사항을 개발자끼리 검토하고 승인하는가? 자신도 기여 이상의 역할을 하고 있는지? 중간 관리자가 있어서 그들에게 먼저 알려야 하는가?

세 번째 변수는 접근 권한이다. '프로젝트에 쓰기 권한이 있어서 직접 쓸 수 있는가? 아니면 읽기만 가능한가?'에 따라서 프로젝트에 기여하는 방식이 매우 달라진다. 쓰기 권한이 없다면 어떻게 수정 사항을 프로젝트에 반영할 수 있을까? 수정사항을 적용하는 정책이 프로젝트에 있는가? 얼마나 많은 시간을 프로젝트에 할애하는가? 얼마나 자주 기여하는가?

이런 질문에 따라 프로젝트에 기여하는 방법과 Workflow 등이 달라진다. 간단한 것부터 복잡한 것까지 각 상황을 살펴보면 실제 프로젝트에 필요한 방식을 선택할 수 있게 된다.

5.2.1 커밋 가이드라인

다른 것보다 먼저 커밋 메시지에 대한 주의사항을 알아보자. 커밋 메시지를 잘 작성하는 가이드라인을 알아두면 다른 개발자와 함께 일하는 데 도움이 많이 된다. Git 프로젝트에 보면 커밋 메시지를 작성하는데 참고할 만한 좋은 팁이 많다. Git 프로젝트의 'Documentation/SubmittingPatches' 문서를 참고하자.

공백문자를 깨끗하게 정리하고 커밋해야 한다. Git은 공백문자를 검사해볼 수 있는 간단한 명령을 제공한다. 커밋을 하기 전에 `git diff --check` 명령으로 공백문자에 대한 오류를 확인할 수 있다. 아래 예제를 보면 잘 못 사용한 공백을 'X' 문자로 바꾸어 표시해준다:

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+    def command(git_cmd)XXXX
```

커밋을 하기 전에 공백문자에 대해 검사를 하면 공백으로 불필요하게 커밋되는 것을 막고 이런 커밋으로 인해 불필요하게 다른 개발자들이 신경 쓰는 일을 방지할 수 있다.

그리고 각 커밋은 논리적으로 구분되는 Changeset이다. 최대한 수정사항을 하나의 주제로 요약할 수 있어야 하고 여러 가지 이슈에 대한 수정사항을 하나의 커밋에 담지 않아야 한다. 여러 가지 이슈를 한꺼번에 수정했다고 하더라도 Staging Area을 이용하여 한 커밋에 하나의 이슈만 담기도록 한다. 작업 내용을 분할하고, 각 커밋마다 적절한 메시지를 작성한다. 같은 파일의 다른 부분을 수정하는 경우에는

`git add -patch` 명령을 써서 한 부분씩 나누어 Staging Area에 저장해야 한다(관련 내용은 6장에서 다룬다). 결과적으로 최종 프로젝트의 모습은 한 번에 커밋을 하든 다섯 번에 나누어 커밋을 하든 똑같다. 하지만, 여러 번 나누어 커밋하는 것이 좋다. 다른 동료가 수정한 부분을 확인할 때나 각 커밋의 시점으로 복원해서 검토할 때 이해하기 훨씬 쉽다. 6장에서 이미 저장된 커밋을 다시 수정하거나 파일을 단계적으로 Staging Area에 저장하는 방법을 살펴본다. 다양한 도구를 이용해서 간단하고 이해하기 쉬운 커밋을 쌓아가야 한다.

마지막으로 명심해야 할 점은 커밋 메시지 자체다. 좋은 커밋 메시지를 작성하는 습관은 Git을 사용하는 데 도움이 많이 된다. 일반적으로 커밋 메시지를 작성할 때 사용하는 규칙이 있다. 메시지의 첫 줄에 50자가 넘지 않는 아주 간략한 메시지를 적어 해당 커밋을 요약한다. 다음 한 줄은 비우고 그다음 줄부터 커밋을 자세히 설명한다. 예를 들어 Git 개발 프로젝트에서는 개발 동기와 구현 상황의 제약조건이나 상황 등을 자세하게 요구한다. 이런 점은 따를 만한 좋은 가이드라인이다. 그리고 현재형 표현을 사용하는 것이 좋다. 예를 들어 "I added tests for (테스트를 추가함)" 보다는 "Add tests for (테스트 추가)" 와 같은 메시지를 작성한다. 아래 예제는 Pope at tpope.net이 작성한 커밋 메시지이다.

```
영문 50글자 이하의 간략한 수정 요약

자세한 설명. 영문 72글자 이상이 되면 줄 바꿈을 하고 이어지는 내용을
작성한다. 특정 상황에서는 첫 번째 줄이 이메일 메시지의 제목이 되고
나머지는 메일 내용이 된다. 간략하게 요약하고 넣는 빈 줄은 자세한
설명을 아예 쓰지 않는 한 매우 중요하다.

이어지는 내용도 한 줄 띄우고 쓴다.

 - 목록 표시도 사용할 수 있다.

 - 보통 '-' 나 '*' 표시를 사용해서 목록을 표현하고 표시 앞에 공백
   하나, 각 목록 사이에는 빈 줄을 하나를 넣는데 상황에 따라 다르다.
```

메시지를 이렇게 작성하면 함께 일하는 사람은 물론이고 자신에게도 매우 유용하다. Git 개발 프로젝트에는 잘 쓰인 커밋 메시지가 많으므로 프로젝트를 내려받아서 `git log --no-merges` 명령으로 꼭 살펴보기를 권한다.

이 책에서 설명하는 예제의 커밋 메시지는 시간 관계상 위와 같이 아주 멋지게 쓰지 않았다. `git commit` 명령에서 `-m` 옵션을 사용하여 간단하게 적는다. 하지만! 저자처럼 하지 말고 시키는 대로 하셔야 한다.

5.2.2 비공개 소규모 팀

두세 명으로 이루어진 비공개 프로젝트가 가장 간단한 프로젝트일 것이다. 비공개라고 함은 소스코드가 공개되지 않은 것을 말하는 것이지 외부에서 접근할 수 없는 것을 말하지 않는다. 모든 개발자는 공유하는 저장소에 쓰기 권한이 있어야 한다.

이런 환경에서는 보통 Subversion 같은 중앙집중형 버전 관리 시스템에서 사용하던 방식을 사용한다. 물론 Git이 가진 오프라인 커밋 기능이나 브랜치 Merge 기능을 이용하긴 하지만 크게 다르지 않다. 가장 큰 차이점은 서버가 아닌 클라이언트 쪽에서 Merge한다는 점이다. 두 개발자가 저장소를 공유하는 시나리오를 살펴보자. 개발자 John씨는 저장소를 Clone하고 파일을 수정하고 나서 로컬에 커밋한다(Git이 출력하는 메시지는 ...으로 줄이고 생략한다).

```
# John's Machine
$ git clone john@githost:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

개발자 Jessica씨도 저장소를 Clone하고 나서 파일을 하나 새로 추가하고 커밋한다:

```
# Jessica's Machine
$ git clone jessica@githost:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Jessica씨는 서버에 커밋을 Push한다:

```
# Jessica's Machine
$ git push origin master
...
To jessica@githost:simplegit.git
   1edee6b..fbff5bc  master -> master
```

John씨도 서버로 커밋을 Push하려고 한다:

```
# John's Machine
$ git push origin master
To john@githost:simplegit.git
 ! [rejected]        master -> master (non-fast forward)
error: failed to push some refs to 'john@githost:simplegit.git'
```

Jessica씨의 Push는 성공했지만, John씨의 커밋은 서버에서 거절된다. Subversion을 사용했던 사람은 이 부분을 이해하는 것이 중요하다. 같은 파일을 수정한 것도 아닌데 왜 Push가 거절되는 걸까?

Subversion에서는 서로 다른 파일을 수정하는 이런 Merge 작업은 자동으로 서버가 처리한다. 하지만, Git은 로컬에서 먼저 Merge해야 한다. John씨는 Push하기 전에 Jessica씨가 수정한 커밋을 Fetch하고 Merge한다:

```
$ git fetch origin
...
From john@githost:simplegit
 + 049d078...fbff5bc master     -> origin/master
```

Fetch하고 나면 John씨의 로컬 저장소는 그림 5-4와 같이 된다.

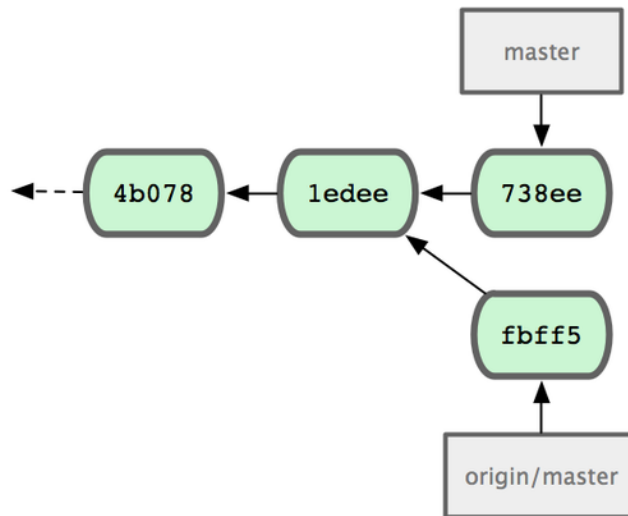



그림 5.4: Fetch하고 난 John씨의 저장소

John씨는 Jessica씨가 저장소로 Push했던 커밋과 를 로컬 저장소에 가져왔다. 하지만, Push하기 전에 Fetch한 브랜치를 Merge해야 한다:

```
$ git merge origin/master
Merge made by recursive.
 TODO |     1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Merge가 잘 이루어지면 John씨의 브랜치는 그림 5-5와 같은 상태가 된다.
John씨는 Merge하고 나서 자신이 작업한 코드가 제대로 동작하는지 확인한다. 그 후에 공유하는 저장소에 Push한다:

```
$ git push origin master
...
To john@githost:simplegit.git
   fbff5bc..72bbc59  master -> master
```

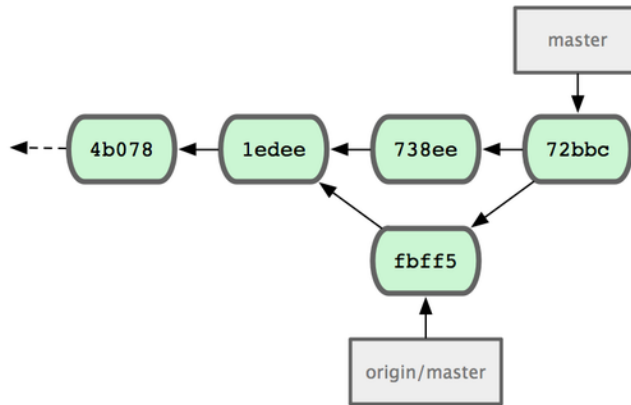



그림 5.5: `origin/master` 브랜치를 Merge하고 난 후, John씨의 저장소

이제 John씨의 저장소는 그림 5-6 처럼 되었다.

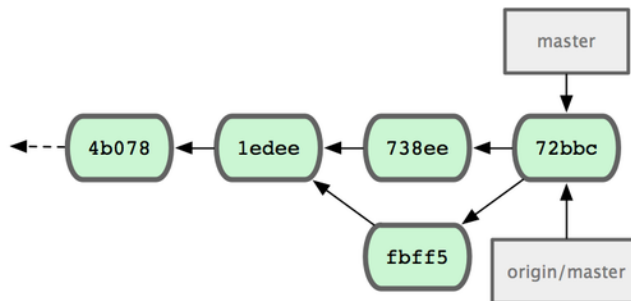



그림 5.6: Push하고 난 후, John씨의 저장소

동시에 Jessica씨는 토픽 브랜치를 하나 만든다. issue54 브랜치를 만들고 세 번에 걸쳐서 커밋한다. 아직 John씨의 커밋을 Fetch하지 않은 상황이기 때문에 그림 5-7과 같은 상황이 된다.

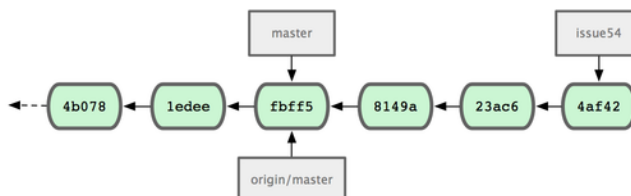



그림 5.7: Jessica씨의 저장소

Jessica씨는 John씨의 작업을 적용하려면 Fetch를 해야 한다:

```
# Jessica's Machine
$ git fetch origin
...
From jessica@githost:simplegit
   fbff5bc..72bbc59  master     -> origin/master
```

위 명령으로 John씨가 Push한 커밋을 모두 내려받는다. 그러면 Jessica씨의 저장소는 그림 5-8과 같은 상태가 된다.

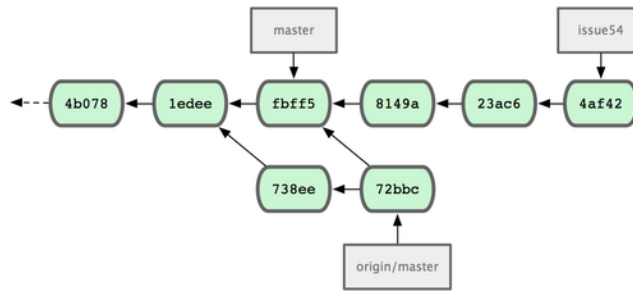



그림 5.8: John씨의 커밋을 Fetch한 후 Jessica씨의 저장소

이제 orgin/master와 Merge할 차례다. Jessica씨는 토픽 브랜치 작업을 마치고 어떤 내용이 Merge 되는지 `git log` 명령으로 확인한다:

```
$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

Merge할 내용을 확인한 Jessica씨는 자신이 작업한 내용과 John씨가 Push한 작업(`origin/master`)을 `master` 브랜치에 Merge하고 Push한다. 모든 내용을 합치기 전에 우선 `master` 브랜치를 Checkout 한다:

```
$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

`origin/master`, `issue54` 모두 `master`보다 Fast-forward된 브랜치이기 때문에 둘 중에 무엇을 먼저 Merge하든 상관이 없다. 물론 어떤 것을 먼저 Merge하느냐에 따라 히스토리 순서는 달라지지만, 최종 결과는 똑같다. Jessica씨는 먼저 `issue54` 브랜치를 Merge한다:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README           |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

보다시피 Fast-forward Merge이기 때문에 별 문제 없이 실행된다. 다음은 John씨의 커밋(`origin/master`)을 Merge한다:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

위와 같이 Merge가 잘 되면 그림 5-9와 같은 상태가 된다.

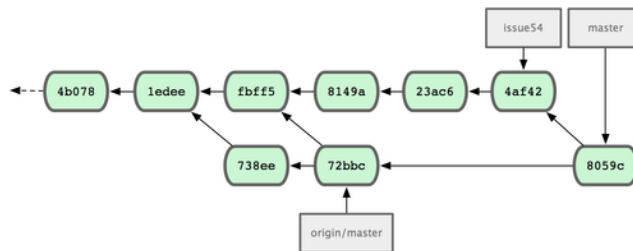



그림 5.9: Merge 이후 Jessica씨의 저장소

`origin/master` 브랜치가 Jessica씨의 `master` 브랜치로 나아갈(reachable) 수 있기 때문에 Push는 성공한다(물론 John씨가 그 사이에 Push를 하지 않았다면):

```
$ git push origin master
...
To jessica@githost:simplegit.git
   72bbc59..8059c15  master -> master
```

두 개발자의 커밋과 Merge가 성공적으로 이루어지고 난 후의 결과는 5-10과 같다.

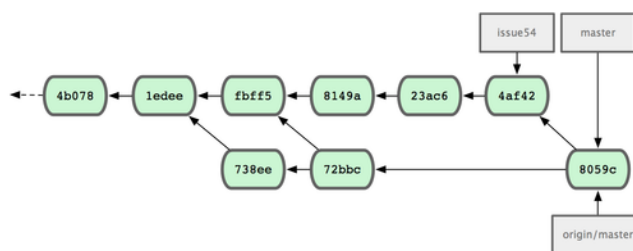



그림 5.10: Jessica씨가 서버로 Push하고 난 후의 저장소

여기서 살펴본 예제가 가장 간단한 상황이다. 토픽 브랜치에서 수정하고 로컬의 `master` 브랜치에 Merge한다. 작업한 내용을 프로젝트의 공유 저장소에 Push하고자 할 때에는 우선 `origin/master` 브랜치를 Fetch하고 Merge한다. 그리고 나서 Merge한 결과를 다시 서버로 Push한다. 이런 Workflow가 일반적이고 그림 5-11로 나타낼 수 있다.

5.2.3 비공개 대규모 팀

이제 비공개 대규모 팀에서의 역할을 살펴보자. 이런 상황에는 보통 팀을 여러 개로 나눈다. 그래서 각각의 작은 팀이 서로 어떻게 하나로 Merge하는지를 살펴본다.

그림 5.11: 여러 개발자가 Git을 사용하는 Workflow

John씨와 Jessica씨는 한팀이고 프로젝트에서 어떤 한 부분을 담당한다. 또한, Jessica씨와 Josie씨도 다른 부분을 담당하는 한 팀이다. 이런 상황이라면 회사는 Integration-manager Workflow를 선택하는 게 좋다. 작은 팀이 수행한 결과물은 Integration-Manager가 Merge하고 공유 저장소의 master 브랜치를 업데이트한다. 팀마다 브랜치를 하나씩 만들고 Integration-Manager는 그 브랜치를 Pull해서 Merge한다.

두 팀에 모두 속한 Jessica씨의 작업 순서를 살펴보자. 우선 Jessica씨는 저장소를 Clone하고 featureA 작업을 먼저 한다. featureA 브랜치를 만들고 수정을 하고 커밋을 한다:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch "featureA"
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

이 수정한 부분을 John씨와 공유해야 한다. 공유하려면 우선 featureA 브랜치를 서버로 Push한다. Integration-Manager만 master 브랜치를 업데이트할 수 있기 때문에 master 브랜치로 Push를 할 수 없고 다른 브랜치로 John과 공유한다:

```
$ git push origin featureA
...
```

```
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica씨는 자신이 한 일을 featureA라는 브랜치로 Push했다는 이메일을 John씨에게 보낸다. John씨의 피드백을 기다리는 동안 Jessica씨는 Josie씨와 함께 하는 featureB 작업을 하기로 한다. 서버의 master 브랜치를 기반으로 새로운 브랜치를 하나 만든다:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

몇 가지 작업을 하고 featureB 브랜치에 커밋한다:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

그럼 Jessica씨의 저장소는 그림 5-12과 같다.

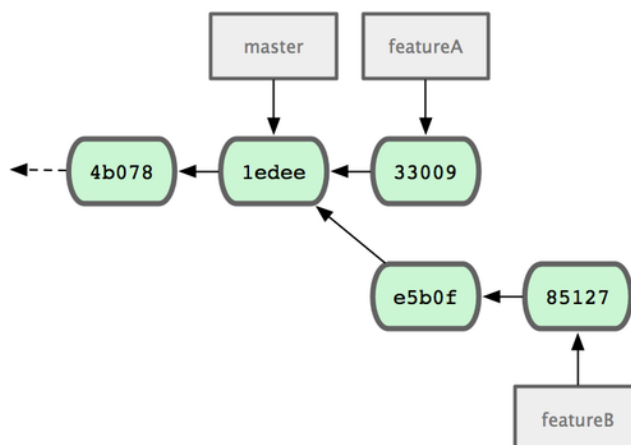



그림 5.12: Jessica씨의 저장소

작업을 마치고 Push하려고 하는데 Jesie씨가 이미 일부 작업을 하고 서버에 featureBee 브랜치로 Push했다는 이메일을 보내왔다. Jessica씨는 Jesie씨의 작업을 먼저 Merge해야만 Push할 수 있다. Merge하기 위해서 우선 `git fetch`로 Fetch한다:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Fetch해 온 브랜치를 `git merge` 명령으로 Merge한다:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Push하려고 하는데 작은 문제가 생겼다. Jessica씨는 featureB 브랜치에서 작업을 했는데 서버에는 브랜치가 featureBee라는 이름으로 되어 있다. 그래서 `git push` 명령으로 Push할 때 로컬 브랜치 featureB 뒤에 콜론(:)과 함께 서버 브랜치 이름을 직접 지정해 준다:

```
$ git push origin featureB:featureBee
...
To jessica@githost:simplegit.git
   fba9af8..cd685d1  featureB -> featureBee
```

이것은 refspec 이란 것을 사용하는 것인데 9장에서 자세하게 설명한다.

John씨가 몇 가지 작업을 하고 나서 featureA에 Push했고 확인해 달라는 내용의 이메일을 보내왔다. Jessica씨는 `git fetch`로 Push한 작업을 Fetch한다:

```
$ git fetch origin
...
From jessica@githost:simplegit
   3300904..aad881d  featureA   -> origin/featureA
```

어떤 것이 업데이트됐는지 `git log`명령으로 확인한다:

```
$ git log origin/featureA ^featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

    changed log output to 30 from 25
```

확인을 마치면 로컬의 featureA 브랜치로 Merge한다:

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb |   10 +++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica씨는 일부 수정하고, 수정한 내용을 다시 서버로 Push한다:

```
$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push origin featureA
...
To jessica@githost:simplegit.git
   3300904..774b3ed  featureA -> featureA
```

위와 같은 작업을 마치고 나면 Jessica씨의 저장소는 그림 5-13과 같은 모습이 된다.

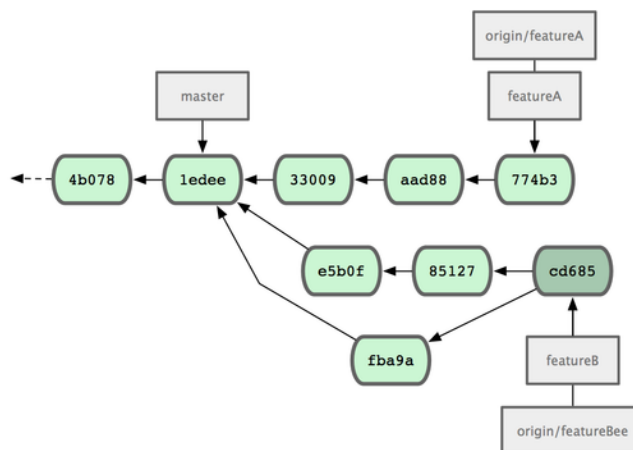



그림 5.13: 마지막 Push하고 난 후의 Jessica씨의 저장소

그럼 featureA와 featureBee 브랜치가 프로젝트의 메인 브랜치로 Merge할 준비가 되었다고 Integration-Manager에게 알려준다. Integration-Manager가 두 브랜치를 모두 Merge하고 난 후에 메인 브랜치를 Fetch하면 그림 5-14와 같은 모양이 된다.

수많은 팀의 작업을 동시에 진행하고 나중에 Merge하는 기능을 사용하려고 다른 버전 관리 시스템에서 Git으로 바꾸는 조직들이 많아지고 있다. 팀은 자신의 브랜치로 작업하지만, 메인 브랜치에 영향을 끼치지 않는다는 점이 Git의 장점이다. 그림 5-15는 이런 Workflow을 나타내고 있다.

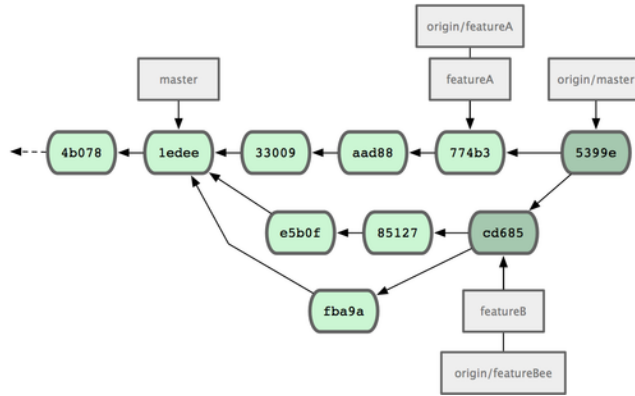



그림 5.14: 두 브랜치가 메인 브랜치에 Merge된 후의 저장소

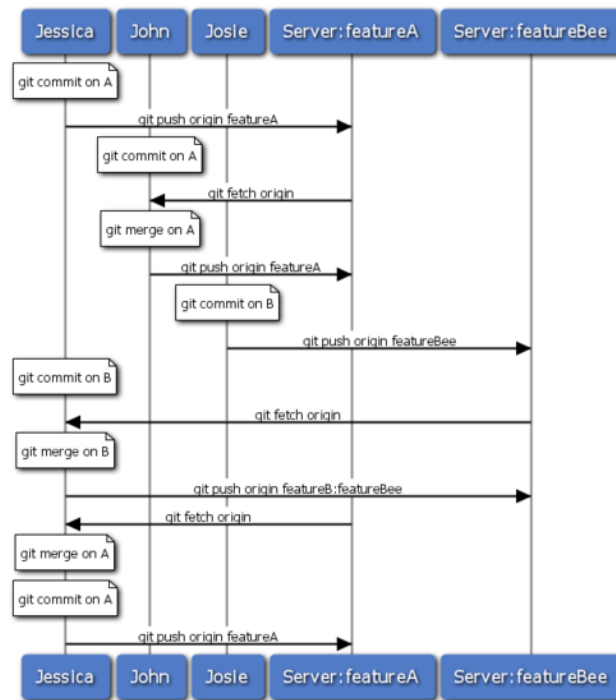



그림 5.15: 대규모 팀의 Workflow

5.2.4 공개 소규모 팀

비공개 팀을 운영하는 것과 공개 팀을 운영하는 것은 약간 다르다. 공개 팀을 운영할 때에는 모든 개발자가 프로젝트의 공유 저장소에 직접적으로 쓰기 권한을 가지지는 않는다. 그래서 프로젝트의 관리자는 몇 가지 일을 더 해줘야 한다. Fork를 지원하는 Git 호스팅에서 Fork를 통해 프로젝트에 기여하는 법을 예제를 통해 살펴본다. repo.or.cz나 Github 같은 Git 호스팅 사이트는 Fork 기능을 지원하며 프로젝트 관리자는 보통 Fork하는 것으로 프로젝트를 운영한다. 다른 방식으로 이메일과 Patch를 사용하는 방식도 있는데 뒤이어 살펴본다.

우선 처음 할 일은 메인 저장소를 Clone 하는 것이다. 그리고 나서 토픽 브랜치를 만들고 일정 부분 기여한다. 그 순서는 아래와 같다:

```
$ git clone (url)
$ cd project
```

```
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
```

`rebase -i` 명령을 사용하면 여러 커밋을 하나의 커밋으로 합치거나 프로젝트의 관리자가 수정사항을 쉽게 이해하도록 커밋을 정리할 수 있다. 6장에서 대화식으로 Rebase하는 방법을 살펴본다.
일단 프로젝트의 웹사이트로 가서 'Fork' 버튼을 누르면 원래 프로젝트 저장소에서 갈라져 나온, 쓰기 권한이 있는 저장소가 하나 만들어진다. 그러면 로컬에서 수정한 커밋을 외부에 있는 그 저장소에 Push 할 수 있다. 그 저장소를 로컬 저장소의 리모트 저장소로 등록한다. 예를 들어 myfork로 등록한다:

```
$ git remote add myfork (url)
```

자 이제 등록한 리모트 저장소에 Push한다. 작업하던 것을 로컬 저장소의 master 브랜치에 Merge 한 후 Push하는 것보다 리모트 브랜치에 바로 Push를 하는 방식이 훨씬 간단하다. 이렇게 하는 이유는 관리자가 토픽 브랜치를 프로젝트에 포함시키고 싶지 않을 때 토픽 브랜치를 Merge하기 이전 상태로 master 브랜치를 되돌릴 필요가 없기 때문이다. 관리자가 토픽 브랜치를 Merge하든 Rebase하든 cherry-pick하든지 간에 결국 다시 관리자의 저장소를 Pull할 때에는 토픽 브랜치의 내용이 들어 있을 것이다:

```
$ git push myfork featureA
```

Fork한 저장소에 Push하고 나면 프로젝트 관리자에게 이 내용을 알려야 한다. 이것을 'Pull Request'라고 한다. git 호스팅 사이트에서 관리자에게 보낼 메시지를 생성하거나 `git request-pull` 명령으로 이메일을 수동으로 만들 수 있다. GitHub의 "pull request" 버튼은 자동으로 메시지를 만들어 준다.
request-pull 명령은 아규먼트를 두 개 입력받는다. 첫 번째 아규먼트는 작업한 토픽 브랜치의 Base 브랜치이다. 두 번째는 토픽 브랜치가 위치한 저장소 URL인데 위에서 등록한 리모트 저장소 이름을 적을 수 있다. 이 명령은 토픽 브랜치 수정사항을 요약한 내용을 결과로 보여준다. 예를 들어 Jessica씨가 John씨에게 Pull 요청을 보내는 상황을 살펴보자. Jessica씨는 토픽 브랜치에 두 번 커밋을 하고 Fork 한 저장소에 Push했다. 그리고 아래와 같이 실행한다:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John Smith (1):
        added a new function

are available in the git repository at:
```

```
  git://githost/simplegit.git featureA

Jessica Smith (2):
      add limit to log function
      change log output to 30 from 25

 lib/simplegit.rb |   10 +++++++++-
 1 files changed, 9 insertions(+), 1 deletions(-)
```

관리자에게 이 내용을 보낸다. 이 내용에는 토픽 브랜치가 어느 시점에 갈라져 나온 것인지, 어떤 커밋이 있는 지, Pull하려면 어떤 저장소에 접근해야 하는지에 대한 내용이 들어 있다.

프로젝트 관리자가 아니라고 해도 보통 origin/master를 추적하는 master 브랜치는 가지고 있다. 그래도 토픽 브랜치를 만들고 일을 하면 관리자가 수정 내용을 거부할 때 쉽게 버릴 수 있다. 토픽 별로 브랜치를 분리해서 일을 하면 그동안 주 저장소의 master 브랜치가 수정됐기 때문에 깨끗한 커밋을 남길 수 없을 수도 있지만 Rebase로 깨끗하게 Merge할 수 있다. 그리고 토픽 브랜치를 새로 만들 때 앞서 Push한 토픽 브랜치에서 시작하지 말고 주 저장소의 master 브랜치로부터 만들어야 한다:

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
$ (email maintainer)
$ git fetch origin
```

그림 5-16 처럼 각 토픽은 일종의 실험실이라고 할 수 있다. 각 토픽은 서로 방해하지 않고 독립적으로 수정하고 Rebase할 수 있다:

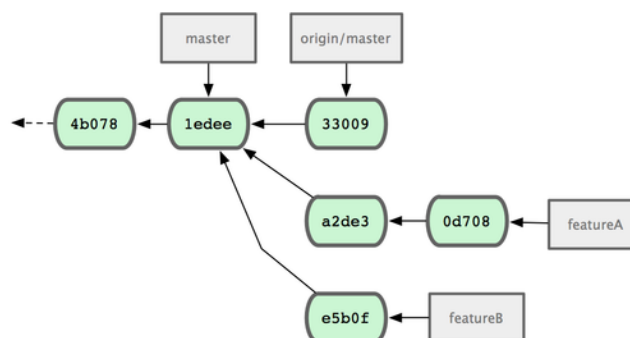



그림 5.16: featureB 수정작업이 끝난 직후 저장소의 모습

프로젝트 관리자가 사람들의 수정사항을 Merge하고 나서 Jessica씨의 브랜치를 Merge하려고 할 때 충돌이 날 수도 있다. 그러면 Jessica씨가 자신의 브랜치를 origin/master에 Rebase해서 충돌을 해결하고 다시 Pull Request을 보낸다:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

위 명령들을 실행하고 나면 그림 5-17과 같아진다.

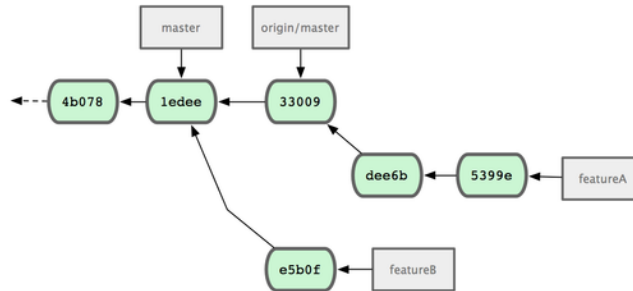



그림 5.17: FeatureA에 대한 Rebase가 적용된 후의 모습

브랜치를 Rebase해 버렸기 때문에 Push할 때 -f 옵션을 주고 강제로 기존에 서버에 있던 브랜치의 내용을 덮어 써야 한다. 아니면 새로운 브랜치를(예를 들어 featureAv2) 서버에 Push해도 된다.
또 다른 시나리오를 하나 더 살펴보자. 프로젝트 관리자는 featureB 브랜치의 내용은 좋지만, 상세 구현은 다르게 하고 싶다. 관리자는 featureB 담당자에게 상세구현을 다르게 해달라고 요청한다. featureB 담당자는 하는 김에 featureB 브랜치를 프로젝트의 최신 master 브랜치 기반으로 옮긴다. 먼저 `origin/master` 브랜치에서 featureBv2 브랜치를 새로 하나 만들고, featureB의 커밋들을 모두 Squash해서 Merge하고, 만약 충돌이 나면 해결하고, 상세 구현을 수정하고, 새 브랜치를 Push한다:

```
$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
```

- squash 옵션은 현재 브랜치에 Merge할 때 해당 브랜치의 커밋을 모두 하나의 커밋으로 합쳐서 Merge한다. - no-commit 옵션을 주면 Git은 Merge하고 나서 자동으로 커밋하지 않는다. 다른 브랜치의 수정사항을 통째로 새로운 브랜치에 Merge하고 나서 좀 더 수정하고 커밋 하나를 새로 만든다.
수정을 마치면 관리자에게 featureBv2 브랜치를 확인해 보라고 메시지를 보낸다 (그림 5-18 참고).

5.2.5 대규모 공개 프로젝트

대규모 프로젝트는 보통 수정사항이나 Patch를 수용하는 자신만의 규칙을 마련해놓고 있다. 프로젝트마다 규칙은 서로 다를 수 있으므로 각 프로젝트의 규칙을 미리 알아둘 필요가 있다. 대규모 프로젝트는 대부분 메일링리스트를 통해서 Patch를 받아들이는데 예제를 통해 살펴본다.
토픽 브랜치를 만들어 수정하는 작업은 앞서 살펴본 바와 거의 비슷하지만, Patch를 제출하는 방식이 다르다. 프로젝트를 Fork 하여 Push하는 것이 아니라 커밋 내용을 메일로 만들어 개발자 메일링리스트에 제출한다:

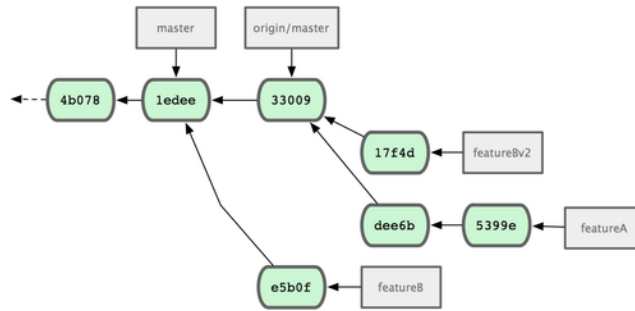



그림 5.18: featureBv2 브랜치를 커밋한 이후 저장소 모습

```
$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit
```

커밋을 두 번 하고 메일링리스트에 보내 보자. `git format-patch` 명령으로 메일링리스트에 보낼 mbox 형식의 파일을 생성한다. 각 커밋은 하나씩 메일 메시지로 생성되는데 커밋 메시지의 첫 번째 줄이 제목이 되고 Merge 메시지 내용과 Patch 자체가 메일 메시지의 본문이 된다. 이 방식은 수신한 이메일에 들어 있는 Patch를 바로 적용할 수 있어서 좋다. 메일 속에는 커밋의 모든 내용이 포함된다. 메일에 포함된 Patch를 적용하는 것은 다음 절에서 살펴본다.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

format-patch 명령을 실행하면 생성한 파일 이름을 보여준다. -M 옵션은 이름이 변경된 파일이 있는지 살펴보라는 옵션이다. 각 파일의 내용은 아래와 같다:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
 lib/simplegit.rb |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
   end

   def ls_tree(treeish = 'master')
--
1.6.2.rc1.20.g8c5b.dirty
```

메일링리스트에 메일을 보내기 전에 각 Patch 메일 파일의 내용을 손으로 고칠 수 있다. --- 줄과 Patch가 시작되는 줄(lib/simplegit.rb로 시작하는 줄) 사이에 내용을 추가하면 개발자는 읽을 수 있지만, 나중에 Patch에 적용되지는 않는다.

특정 메일 프로그램을 사용하거나 이메일을 보내는 명령어로 메일링리스트에 보낼 수 있다. 붙여 넣기로 위의 내용이 그대로 들어가지 않는 메일 프로그램도 있다. 사용자 편의를 위해 공백이나 줄 바꿈 문자 등을 넣어 주는 메일 프로그램은 원본 그대로 들어가지 않는다.

다행히 Git에는 Patch 메일을 그대로 보낼 수 있는 도구가 있다. IMAP 프로토콜로 보낸다. 저자가 사용하는 방법으로 Gmail을 사용하여 Patch 메일을 전송하는 방법을 살펴보자. 추가로 Git 프로젝트의 Documentation/SubmittingPatches 문서의 마지막 부분을 살펴보면 다양한 메일 프로그램으로 메일을 보내는 방법을 설명한다.

메일을 보내려면 먼저 ~/.gitconfig 파일에서 이메일 부분 설정한다. git config 명령으로 추가할 수도 있고 직접 파일을 열어서 추가할 수도 있다. 아무튼, 아래와 같이 설정을 한다:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = p4ssw0rd
  port = 993
  sslverify = false
```

IMAP 서버가 SSL을 사용하지 않으면 마지막 두 줄은 필요 없고 host에서 imaps:// 대신 imap://로 한다. 이렇게 설정하면 git imap-send 명령으로 메일을 전송할 수 있다:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
```

```
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

이후 Gmail의 Draft 폴더로 가서 To 부분을 메일링리스트의 주소로 변경하고 CC 부분에 해당 메일을 참고해야 하는 관리자나 개발자의 메일 주소를 적고 실제로 전송한다.

SMTP 서버로도 패치를 보낼 수 있다. `git config` 명령으로 설정을 하나씩 입력하거나 `~/.gitconfig` 파일의 sendmail 부분을 손으로 직접 수정한다:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

이렇게 했으면 `git send-mail`로 패치를 보낸다:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Email ' s will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Git으로 메일을 보내면 아래와 같은 로그 메시지가 출력된다:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
  \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>
```

```
Result: OK
```

5.2.6 요약

이번 절에서는 다양한 Workflow에 따라 Git을 어떻게 사용하는 지 살펴보고 그에 필요한 도구들을 설명했다. 다음 절에서는 동전의 뒷면인 프로젝트를 운영하는 방법에 대하여 살펴본다. 즉 친절한 Dictator나 Integration-Manager가 되어 보는 것이다.

5.3 프로젝트 운영하기

효율적으로 기여하는 방법뿐만 아니라 효율적으로 운영하는 방법도 알아야 한다. 언젠가는 단순히 프로젝트에 기여하는 것이 아니라 프로젝트를 직접 운영해야 할 수도 있다. 프로젝트를 운영하는 것은 크게 두 가지로 이루어진다. 하나는 format-patch 명령으로 생성한 Patch를 이메일로 받아서 프로젝트에 Patch를 적용하는 것이다. 다른 하나는 프로젝트의 다른 리모트 저장소로부터 변경 내용을 Merge하는 것이다. 저장소를 아주 깔끔하고 정돈된 상태로 운영하고 Patch를 적용하거나 수정사항을 확인하기 쉬운 상태를 유지하려면 좋은 운영 방식을 터득해야 한다. 좋은 운영 방식은 다른 사람들이 이해하기 쉽고 프로젝트가 오랫동안 운영돼도 흐트러짐이 없어야 한다.

5.3.1 토픽 브랜치에서 일하기

메인 브랜치에 통합하기 전에 임시로 토픽 브랜치를 하나 만들고 거기에 통합해 보고 나서 다시 메인 브랜치에 통합하는 것이 좋다. 이렇게 하면 Patch를 적용할 때 이리저리 수정해 보기도 하고 좀 더 고민해 봐야 하면 Patch를 적용해둔 채로 나중으로 미룰 수도 있다. 무슨 Patch인지 브랜치 이름에 간단히 적어주면 다른 작업을 하다가 나중에 이 브랜치로 돌아왔을 때 기억해내기 훨씬 수월하다. 프로젝트의 관리자라면 이런 토픽 브랜치의 이름을 잘 지어야 한다. 예를 들어 sc라는 사람이 작업한 Patch라면 `sc/ruby_client` 처럼 앞에 닉네임을 붙여서 브랜치를 만들 수 있다. master 브랜치에서 새 토픽 브랜치를 아래와 같이 만든다:

```
$ git branch sc/ruby_client master
```

`checkout -b` 명령으로 브랜치를 만들고 Checkout까지 한 번에 할 수 있다:

```
$ git checkout -b sc/ruby_client master
```

이렇게 토픽 브랜치를 만들고 Patch를 적용해보고 적용한 내용을 다시 Long-Running 브랜치로 Merge한다.

5.3.2 이메일로 받은 Patch를 적용하기

이메일로 받은 Patch를 프로젝트에 적용하기 전에 우선 토픽 브랜치에 Patch를 적용해 본다. Patch를 적용하는 방법은 `git apply` 명령을 사용하는 것과 `git am` 명령을 사용하는 것 두 가지가 있다.

apply 명령을 사용하는 방법

`git diff`나 Unix의 diff 명령으로 만든 Patch파일을 적용할 때에는 `git apply` 명령을 사용한다. Patch 파일이 `/tmp/patch-ruby-client.patch`라고 하면 아래와 같은 명령으로 Patch를 적용할 수 있다:

```
$ git apply /tmp/patch-ruby-client.patch
```

위 명령을 실행하면 Patch 파일 내용에 따라 현재 디렉토리의 파일들을 변경한다. 위 명령은 `patch -p1` 명령과 거의 같다. 하지만, 이 명령이 patch 명령보다 훨씬 더 꼼꼼하게 비교한다. `git diff`로 생성한 Patch 파일에 파일을 추가하거나, 파일을 삭제하고, 파일의 이름을 변경하는 내용이 들어 있으면 그대로 적용된다. 이런 것은 patch 명령으로 할 수 없다.
그리고 `git apply`는 "모두 적용, 아니면 모두 취소" 모델을 사용하기 때문에 Patch를 적용하는 데 실패하면 Patch를 적용하기 이전 상태로 전부 되돌려 놓는다. Patch 명령은 여러 파일에 적용하다가 중간에 실패하면 거기서 그대로 중단하기 때문에 깔끔하지 못하다. `git apply`는 Patch보다 훨씬 결벽증적이다. 이 명령은 자동으로 커밋해 주지 않기 때문에 변경된 파일을 직접 Staging Area에 추가하고 커밋해야 한다.
실제로 Patch를 적용하기 전에 Patch가 잘 적용되는지 한 번 시험해보려면 `git apply --check` 명령을 사용한다:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

화면에 아무런 내용도 뜨지 않으면 Patch가 깔끔하게 적용된다는 것이다. 이 명령은 Patch를 적용해 보고 에러가 발생하면 0이 아닌 값을 반환하기 때문에 쉘 스크립트에서도 사용할 수 있다.

am 명령을 사용하는 방법

프로젝트 기여자가 Git의 format-patch 명령을 잘 사용하면 관리자의 작업은 훨씬 쉬워진다. format-patch 명령으로 만든 Patch 파일은 기여자의 정보와 커밋 정보가 포함되어 있기 때문이다. 그래서 기여자가 diff보다 format-patch를 사용하도록 권해야 한다. `git apply`는 기존의 Patch파일에만 사용한다.
format-patch 명령으로 생성한 Patch 파일은 `git am` 명령으로 적용한다. `git am`은 메일 여러 통이 들어 있는 mbox파일을 읽어서 Patch한다. mbox파일은 간단한 텍스트 파일이고 그 내용은 아래와 같다:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20
```

이 내용은 format-patch 명령으로 생성한 파일의 앞부분이다. 이 파일은 mbox 형식이다. 받은 메일이 `git send-email`로 만든 메일이라면 mbox 형식으로 저장하고 이 mbox 파일을 `git am` 명령으로 적용한다. 사용하는 메일 클라이언트가 여러 메일을 하나의 mbox 파일로 저장할 수 있다면 메일 여러 개를 한 번에 Patch할 수 있다.
이메일로 받은 것이 아니라 이슈 트래킹 시스템 같은데 올라온 파일이라면 먼저 내려받고서 `git am` 명령으로 Patch한다:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Patch가 성공하면 자동으로 새로운 커밋이 하나 만들어진다. Patch 파일에 들어 있는 기여자의 이메일, 작성시간, 커밋 메시지를 뽑아서 커밋에 함께 저장한다. 예를 들어 위의 mbox 예제 파일을 적용해서 생성되는 커밋은 아래와 같다:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author:     Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit:     Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

   add limit to log function

   Limit log functionality to the first 20
```

커밋 정보는 누가 언제 Patch했는 지 알려 준다. Author 정보는 실제로 누가 언제 Patch파일을 만들었는지 알려 준다.
Patch에 실패할 수도 있다. 보통 Patch가 생성된 시점보다 해당 브랜치가 너무 업데이트 됐을 때나 아직 적용되지 않은 다른 Patch가 필요한 경우에 일어난다. 이러면 `git am` 명령은 Patch를 중단하고 사용자에게 어떻게 처리할지 물어온다:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
  (충돌을 해결하면 "git am --resolved" 입력)
If you would prefer to skip this patch, instead run "git am --skip".
  (Patch 적용을 생략하려면 "git am --skip" 입력)
To restore the original branch and stop patching run "git am --abort".
  (Patch 적용을 중단하려면 "git am --abort" 입력)
```

성공적으로 Patch하지 못하면 git은 Merge나 Rebase의 경우처럼 문제를 일으킨 파일에 충돌 표시를 해 놓는다. Merge나 Rebase할 때 충돌을 해결하는 것처럼 Patch의 충돌도 해결할 수 있다. 충돌한 파일을 열어서 충돌 부분을 수정하고 나서 Staging Area에 추가하고 `git am --resolved` 명령을 입력한다:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

충돌이 났을 때 Git에게 좀 더 머리를 써서 Patch를 적용하도록 하려면 -3 옵션을 사용한다. 이 옵션은 Git에게 3-way Patch를 적용해 보라고 하는 것이다. Patch가 어느 시점에서 갈라져 나온 것인지 알 수 없기 때문에 이 옵션은 기본적으로 비활성화돼 있다. 하지만, 같은 프로젝트의 커밋이라면 기본옵션보다 훨씬 똑똑하게 충돌 상황을 해결한다.

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

위의 경우는 이미 Patch한 것을 다시 Patch하는 상황이다. -3 옵션이 없었으면 충돌을 알아서 해결하지 못했을 것이다.

하나의 mbox 파일에 들어 있는 여러 Patch를 적용할 때 대화형 방식을 사용할 수 있다. 이 방식을 사용하면 Patch를 적용하기 전에 Patch를 적용할 때마다 묻는다:

```
$ git am -3 -i mbox
Commit Body is:
--------------------------
seeing if this helps the gem
--------------------------
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

이 옵션은 Patch를 여러 개 적용할 때 유용하다. 적용하려는 Patch의 내용을 미리 꼭 기억해두지 않아도 되고 적용하기 전에 이미 적용된 Patch인지 알 수 있다.

모든 Patch를 토픽 브랜치에 적용하고 커밋까지 마치면 Long-Running 브랜치에 어떻게 통합할지를 결정해야 한다.

5.3.3 리모트 브랜치로부터 통합하기

프로젝트 기여자가 자신의 저장소를 만들고 커밋을 몇 번 하고 저장소의 URL과 변경 내용을 메일로 보내왔다면 URL을 리모트 저장소로 등록하고 Merge할 수 있다.
예를 들어 Jessica씨는 ruby-client 브랜치에 엄청난 기능을 만들어 놨다고 메일을 보내왔다. 이 리모트 브랜치를 등록하고 Checkout해서 테스트한다:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

후에 Jessiaca씨가 이메일로 또 다른 엄청난 기능을 개발한 브랜치를 보내오면 이미 저장소를 등록했기 때문에 간단히 Fetch하고 Checkout할 수 있다.
다른 개발자들과 함께 지속적으로 개발할 때는 이 방식이 가장 사용하기 좋다. 물론 기여하는 사람이 간단한 Patch를 이따금씩만 만들어 내면 이메일로 Patch 파일을 받는 것이 낫다. 기여자가 저장소 서버를 만들어 커밋하고 관리자가 리모트 저장소로 등록해서 Patch를 합치는 작업보다 시간과 노력이 덜 든다. 물론 Patch 한두 개를 보내는 사람들까지도 모두 리모트 저장소로 등록해서 사용해도 된다. 스크립트나 호스팅 서비스를 사용하면 좀 더 쉽게 관리할 수 있다. 어쨌든 어떤 방식이 좋을지는 우리가 어떻게 개발하고 어떻게 기여할지에 달렸다.
리모트 저장소로 등록하면 커밋의 히스토리도 알 수 있다. Merge할 때 어디서 부터 커밋이 갈라졌는지 알 수 있기 때문에 -3 옵션을 주지 않아도 자동으로 3-way Merge가 적용된다.
리모트 저장소로 등록하지 않고도 Merge할 수 있다. 계속 함께 일할 개발자가 아닐 때 사용하면 좋다. 아래는 리모트 저장소로 등록하지 않고 URL을 직접 사용하여 Merge를 하는 예이다:

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch            HEAD       -> FETCH_HEAD
Merge made by recursive.
```

5.3.4 무슨 내용인지 확인하기

기여물이 포함된 토픽 브랜치가 있으니 이제 그 기여물을 Merge할지 말지 결정야 한다. 이번 절에서는 메인 브랜치에 Merge할 때 필요한 명령어를 살펴본다. 주로 토픽 브랜치를 검토하는데 필요한 명령이다.
먼저 지금 작업하는 브랜치에서 master 브랜치에 속하지 않는 커밋만 살펴보는 것이 좋다. - not 옵션으로 히스토리에서 master 브랜치에 속한 커밋은 제외하고 살펴본다. 예를 들어 contrib 브랜치에 Patch를 두 개 Merge했으면 아래와 같은 명령어로 그 결과를 살펴볼 수 있다:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700
```

```
    seeing if this helps the gem

commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700

    updated the gemspec to hopefully work better
```

`git log` 명령에 -p 옵션을 주면 각 커밋에서 실제로 무슨 내용이 변경됐는지 살펴볼 수 있다. 이 옵션은 각 commit의 뒤에 diff의 내용을 출력해 준다.

토픽 브랜치를 다른 브랜치에 Merge하기 전에 어떤 부분이 변경될지 미리 살펴볼 수 있다. 이때는 색다른 명령을 사용해야 한다. 물론 아래와 같은 명령을 사용할 수도 있다:

```
$ git diff master
```

이 명령은 diff 내용을 보여주긴 하지만 잘못된 것을 보여줄 수도 있다. 토픽 브랜치에서 작업하는 동안 master 브랜치에 새로운 커밋이 추가될 수도 있다. 그래서 기대하는 diff 결과가 아닐 수 있다. 지금 이 명령은 각 브랜치의 마지막 스냅샷을 비교한다. master 브랜치에 한 줄을 추가되면 토픽 브랜치에서 한 줄 삭제한 것으로 보여 준다.

master 브랜치가 가리키는 커밋이 토픽 브랜치의 조상이라면 아무 문제 없다. 하지만, 그렇지 않은 경우라면 이 diff 도구는 토픽 브랜치에만 있는 내용은 추가하는 것이고 master 브랜치에만 있는 내용은 삭제하는 것으로 간주한다.

정말 보고 싶은 것은 토픽 브랜치에 추가한 것이고 결국에는 이것을 master 브랜치에 추가하려는 것이다. 그러니까 master 브랜치와 토픽 브랜치의 공통 조상인 커밋을 찾아서 토픽 브랜치가 현재 가리키는 커밋과 비교해야 한다.

아래와 같은 명령으로 공통 조상인 커밋을 찾고 이 조상 커밋에서 변경된 내용을 살펴본다:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

이 방법으로 원하는 결과를 얻을 수 있지만, 사용법이 불편하다. Git은 Triple-Dot으로 간단하게 위와 같이 비교하는 방법을 지원한다. diff 명령을 사용할 때 두 브랜치 사이에 …를 쓰면, 두 브랜치의 공통 조상과 브랜치의 마지막 커밋을 비교한다:

```
$ git diff master...contrib
```

이 명령은 master 브랜치로부터 현재 토픽 브랜치의 다른 것들만 보여주기 때문에 기억해두면 매우 유용하게 사용할 수 있을 것이다.

5.3.5 기여물 통합하기

기여물을 토픽 브랜치에 다 적용하고 Long-Running 브랜치나 master 브랜치로 통합할 준비가 되었다면 이제 어떻게 해야 할까? 프로젝트를 운영하는 데 쓰는 작업 방식은 어떤 것이 있을까? 앞으로 그 예제 몇 가지를 살펴본다.

Merge Workflow

바로 master 브랜치에 Merge하는 것이 가장 간단하다. 이 Workflow에서는 master 브랜치가 안전한 코드라고 가정한다. 토픽 브랜치를 검증하고 master 브랜치로 Merge할 때마다 토픽 브랜치를 삭제한다. 그림 5-19처럼 `ruby_client` 브랜치와 `php_client` 브랜치가 있을 때 `ruby_client` 브랜치를 master 브랜치로 Merge한 후 `php_client` 브랜치를 Merge하면 그림 5-20과 같아진다:

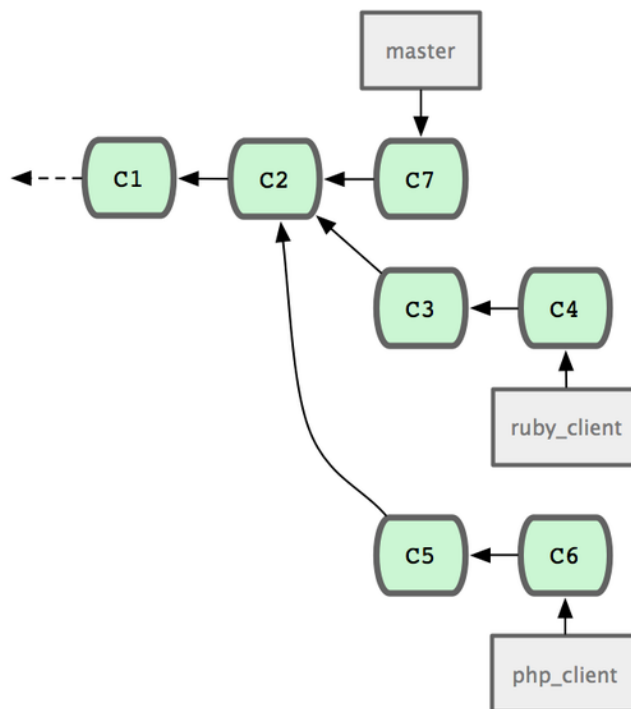



그림 5.19: 저장소의 두 브랜치

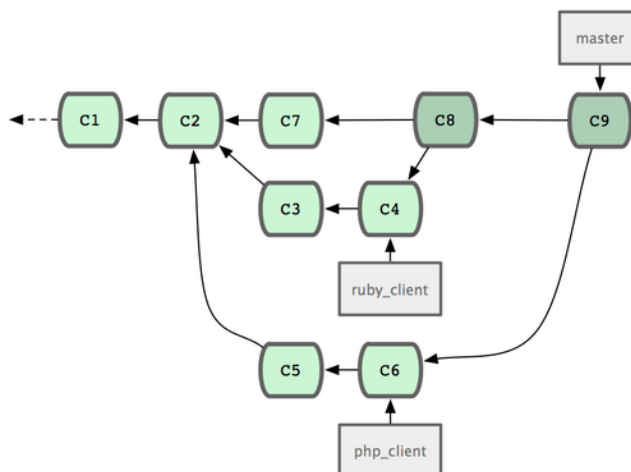



그림 5.20: Merge한 후의 저장소

이 Workflow은 간단하지만, 프로젝트의 규모가 커지면 문제가 생길 수 있다.

개발자가 많고 규모가 큰 프로젝트에서는 두 단계로 Merge하는 것이 좋다. 그래서 Long-Running 브랜치를 두 개로 유지해야 한다. master 브랜치는 아주 안정적인 버전을 릴리즈하기 위해서 사용한다. develop 브랜치는 새로 수정된 코드를 통합할 때 사용한다. 그리고 두 브랜치를 모두 저장소에 Push한다. 우선 develop 브랜치에 토픽 브랜치(그림 5-21)를 그림 5-22과 같이 Merge한다. 그 후에 릴리즈해도 될만한 수준이 되면 master 브랜치를 develop 브랜치까지 Fast-forward시킨다(그림 5-23).

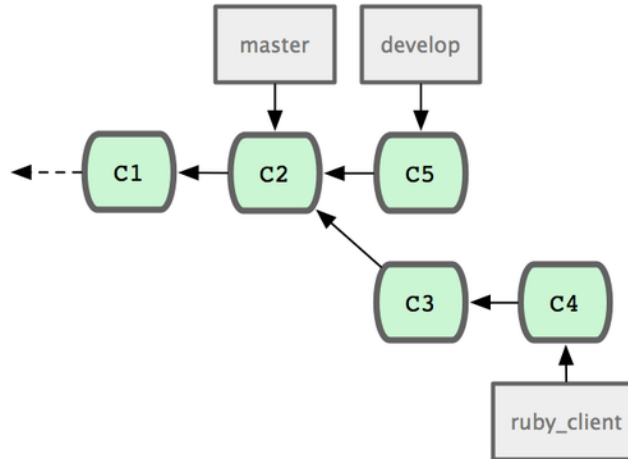



그림 5.21: 토픽 브랜치를 Merge하기 전

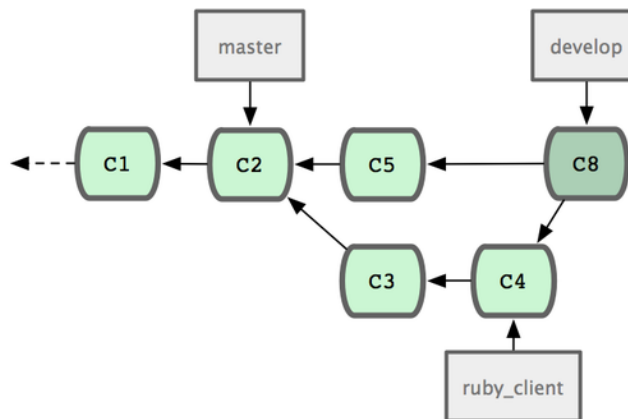



그림 5.22: 토픽 브랜치를 Merge한 후

이 Workflow을 사용하면 프로젝트 저장소를 Clone하고 나서 개발자가 안정 버전이 필요하면 master 브랜치를 빌드하고 안정적이지 않더라도 좀 더 최신 버전이 필요하면 develop 브랜치를 Checkout하여 빌드한다. 이 개념을 좀 더 확장해서 사용할 수 있다. 토픽 브랜치를 검증하기 위한 integrate 브랜치를 만들어 Merge하고 토픽 브랜치가 검증되면 develop 브랜치에 머지한다. 그리고 develop 브랜치에서 충분히 안정하다는 것이 증명되면 그때 master 브랜치에 Merge한다.

대규모 Merge Workflow

Git을 개발하는 프로젝트는 Long-Running의 브랜치를 4개 운영한다. 각 브랜치 이름은 master, next, pu (Proposed Updates), maint 이다. maint는 마지막으로 릴리즈한 버전을 지원하는 브랜치다. 기여자가 새로운 기능을 제안하면 관리자는 그림 5-24처럼 자신의 저장소에 토픽 브랜치를 만

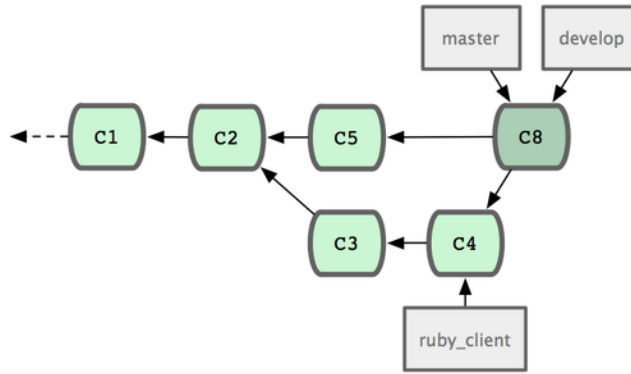



그림 5.23: 토픽 브랜치를 릴리즈한 후

들어 관리한다. 그리고 토픽에 부족한 점은 없는지, 안정적인지 계속 테스트한다. 안정화되면 next로 Merge하고 저장소에 Push한다. 그러면 모두가 잘 통합됐는지 확인할 수 있다.

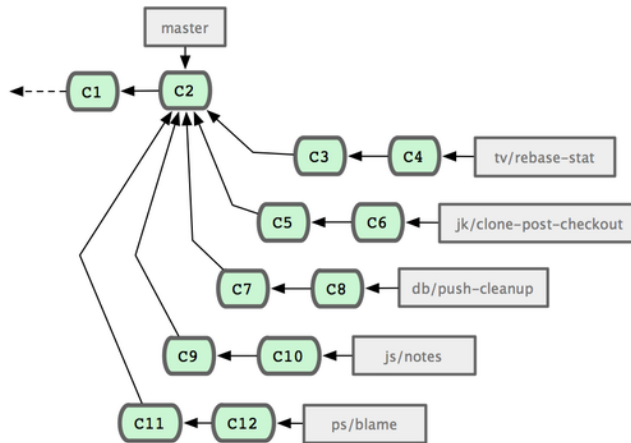



그림 5.24: 토픽 브랜치를 동시에 여러 개 관리하는 것은 복잡하다

토픽 브랜치가 좀 더 개선돼야 하면 next가 아니라 pu에 Merge한다. 그 후에 충분히 검증을 마치면 pu에서 next로 옮기고 next를 기반으로 pu를 다시 만든다. next에는 아직 master에 넣기에 모자라 보이는 것들이 들어 있다. 즉 next 브랜치는 정말 가끔 Rebase하고 pu는 자주 Rebase하지만 master는 항상 Fast-forward한다(그림 5-25).

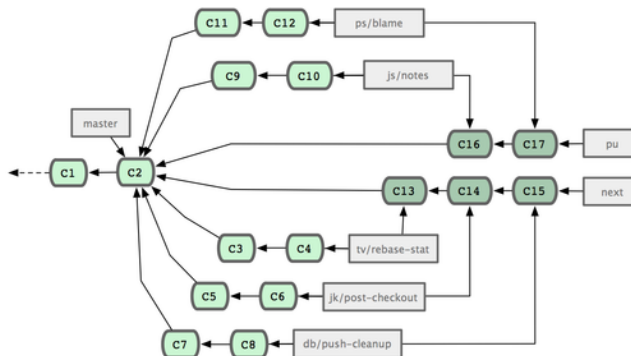



그림 5.25: 토픽 브랜치를 Long-Running 브랜치로 Merge하기

토픽 브랜치가 결국 master 브랜치로 Merge되면 저장소에서 삭제한다. 그리고 이전 릴리즈 버전에 Patch가 필요하면 maint 브랜치를 이용해 대응한다. Git을 개발하는 프로젝트를 Clone하면 브랜치가

4개 있고 각 브랜치를 이용하여 진행사항을 확인해볼 수 있다. 그래서 새로운 기능을 추가하려면 적당한 브랜치를 보고 고른다. 이 Workflow는 잘 구조화돼 있어서 코드가 새로 추가돼도 테스트하기 쉽다.

Rebase와 Cherry-Pick Workflow

히스토리를 평평하게 관리하려고 Merge보다 Rebase나 Cherry-Pick을 더 선호하는 관리자들도 있다. 토픽 브랜치에서 작업을 마친 후 master에 통합할 때 master 브랜치를 기반으로 Rebase한다. 그러면 커밋이 다시 만들어 진다. master 대신 develop 등의 브랜치에도 가능하다. 문제가 없으면 master 브랜치를 Fast-forward시킨다. 이렇게 평평한 히스토리를 유지할 수 있다.
한 브랜치에서 다른 브랜치로 작업한 내용을 옮기는 또 다른 방식으로 Cherry-pick이란 것도 있다. Git의 Cherry-pick은 커밋 하나만 Rebase하는 것이다. 커밋 하나로 Patch 내용을 만들어 현재 브랜치에 적용을 하는 것이다. 토픽 브랜치에 있는 커밋중에서 하나만 고르거나 토픽 브랜치에 커밋이 하나밖에 없을 때 Rebase보다 유용하다. 그림 5-26의 예를 들어보자.

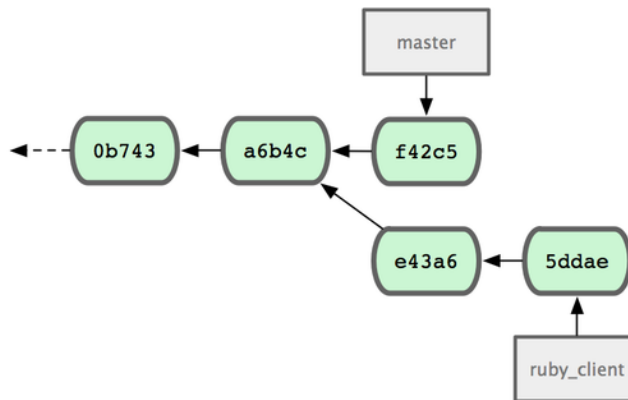



그림 5.26: Cherry-pick을 실행하기 전의 저장소

e43a6 커밋 하나만 현재 브랜치에 적용하려면 아래와 같은 명령을 실행한다:

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

위 명령을 실행하면 e43a6 커밋에서 변경된 내용을 현재 브랜치에 똑같이 적용을 한다. 하지만, 변경을 적용한 시점이 다르므로 새 커밋의 SHA-1 해시 값은 달라진다. 명령을 실행하고 나면 그림 5-27과 같이 될 것이다.
Rebase나 Cherry-pick 방식으로 토픽 브랜치를 합치고 나면 필요없는 토픽 브랜치나 커밋은 삭제한다.

5.3.6 릴리즈 버전에 태그 달기

적당한 때가 되면 릴리즈해야 한다. 그리고 언제든지 그 시점으로 되돌릴 수 있게 태그를 다는 것이 좋다. 2장에서 살펴본 대로 태그를 달면 된다. 서명된 태그를 달면 아래와 같이 출력된다:

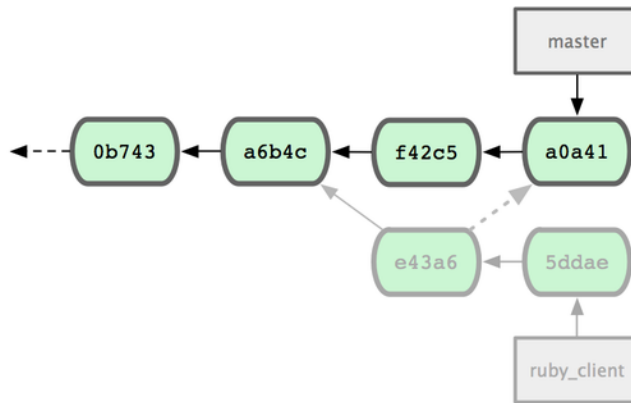



그림 5.27: Cherry-pick 방식으로 커밋 하나를 적용한 후의 저장소

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

태그에 서명하면 서명에 사용한 PGP 공개키도 배포해야 한다. Git 개발 프로젝트는 관리자의 PGP 공개키를 Blob 형식으로 Git 저장소에 함께 배포한다. 이 Blob파일을 사용하여 태그에 서명했다. 아래와 같은 명령으로 어떤 PGP 공개키를 포함할지 확인한다:

```
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
---------------------------------
pub   1024D/F721C45A 2009-02-09 [expires: 2010-02-09]
uid                  Scott Chacon <schacon@gmail.com>
sub   2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

`git hash-object`라는 명령으로 공개키를 바로 Git 저장소에 넣을 수 있다. 이 명령은 Git 저장소 안에 Blob 형식으로 공개키를 저장해주고 그 Blob의 SHA-1 값을 알려준다:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

이 SHA-1 해시 값으로 PGP 공개키를 가리키는 태그를 만든다:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

`git push --tags` 명령으로 앞서 만든 maintainer-pgp-pub 태그를 공유한다. 다른 사람이 태그의 서명을 확인하려면 우선 Git 저장소에 저장된 PGP 공개키를 꺼내서 GPG키 데이터베이스에 저장해야 한다:

```
$ git show maintainer-pgp-pub | gpg --import
```

사람들은 이렇게 공개키를 얻어서 서명된 태그를 확인한다. 또한, 관리자가 태그 메시지에 서명을 확인하는 방법을 적어 놓으면 좋다. `git show <tag>`으로 어떻게 서명된 태그를 확인하는지 설명한다.

5.3.7 빌드넘버 만들기

Git은 'v123' 처럼 숫자 형태로 커밋 이름을 만들지 않기 때문에 사람이 기억하기 어렵다. 하지만 `git describe` 명령으로 좀 더 사람이 기억하기 쉬운 이름을 얻을 수 있다. Git은 가장 가까운 태그의 이름과, 태그에서 얼마나 더 커밋이 쌓였는지, 그리고 해당 커밋의 SHA-1 값을 조금 가져다가 이름을 만든다:

```
$ git describe master
v1.6.2-rc1-20-g8c5b85c
```

이렇게 사람이 읽을 수 있는 이름으로 스냅샷이나 빌드를 만든다. 만약 저장소에서 Clone한 후 소스코드로 Git을 설치하면 `git --version` 명령은 이렇게 생긴 빌드넘버를 보여준다. 태그가 달린 커밋에 `git describe`명령을 사용하면 다른 정보 없이 태그 이름만 사용한다.
`git describe` 명령은 -a나 -s 옵션을 주고 만든 Annotated 태그가 필요하다. 릴리즈 태그는 `git describe` 명령으로 만드니까 꼭 이름이 적당한지 사전에 확인해야 한다. 그리고 이 값은 Checkout이나 Show 명령에도 사용할 수 있지만, 전적으로 이름 뒤에 붙은 SHA-1 값을 사용한다. 그래서 이 값으로는 해당 커밋을 못 찾을 수도 있다. 최근 Linux Kernel은 충돌 때문에 축약된 SHA-1를 8자에서 10자로 늘렸다. 이제는 8자일 때 생성한 값은 사용할 수 없다.

5.3.8 릴리즈 준비하기

먼저 Git을 사용하지 않는 사람을 위해 소스코드 스냅샷을 압축한다. 쉽게 압축할 수 있도록 Git은 `git archive` 명령을 지원한다:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

이 압축 파일을 풀면 프로젝트의 가장 마지막 스냅샷이 나온다. ZIP 형식으로 압축파일을 만들려면 `--format=zip` 옵션을 사용한다:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

이렇게 압축한 스냅샷 파일은 Website나 이메일로 사람들에게 배포할 수 있다.

5.3.9 Shortlog 보기

이메일로 프로젝트의 변경 사항을 사람들에게 알려야 할 때, `git shortlog` 명령을 사용하면 지난 릴리즈 이후의 변경 사항 목록을 쉽게 얻어올 수 있다. `git shortlog` 명령은 주어진 범위에 있는 커밋을 요약해준다. 아래는 최근 릴리즈 버전인 v1.0.1 이후의 커밋을 요약해 주는 예제이다:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
      Add support for annotated tags to Grit::Tag
      Add packed-refs annotated tag support.
      Add Grit::Commit#to_patch
      Update version and History.txt
      Remove stray `puts`
      Make ls_tree ignore nils

Tom Preston-Werner (4):
      fix dates in history
      dynamic version method
      Version bump to 1.0.2
      Regenerated gemspec for version 1.0.2
```

이렇게 Author를 기준으로 정리한 커밋을 이메일로 전송한다.

5.4 요약

이제 Git 프로젝트에 기여하고, 자신의 프로젝트를 운영하고, 다른 사람이 기여한 내용을 통합하는 것 정도는 쉽게 할 수 있을 것이다. 일단 쓸만한 Git 개발자가 된 것을 축하한다. 다음 장에서 복잡한 상황을 다루는 방법과 강력한 도구들을 배우고 나면 Git 장인이라고 불릴 수 있을 것이다.

6장

Git 도구

지금까지 일상적으로 자주 사용하는 명령어들과 몇 가지 Workflow를 배웠다. 파일을 추적하고 커밋하는 등의 기본적인 명령어뿐만 아니라 Staging Area가 왜 좋은지도 배웠고 가볍게 토픽 브랜치를 만들고 Merge하는 방법도 다뤘다. 이제는 소스코드 관리를 Git 저장소로 충분히 해낼 수 있을 것이다.
이 장에서는 일상적으로 사용하지는 않지만 위급한 상황에서 반드시 필요한 Git 도구를 살펴본다.

6.1 리비전 조회하기

리비전 하나를 조회할 수도 있고 범위를 주고 여러 개를 조회할 수도 있다. 잘 쓰진 않지만 알아두는게 좋다.

6.1.1 리비전 하나 가리키기

사람은 커밋을 나타내는 SHA-1 해시 값을 쉽게 기억할 수 없다. 이 절에서는 커밋을 표현하는 방법을 몇 가지 설명한다. 좀 더 사람이 기억하기 쉬운 방법들이다.

6.1.2 짧은 SHA-1

해시 값의 앞 몇 글자만으로도 어떤 커밋인지 충분히 식별할 수 있다. 중복되지 않으면 해시 값의 앞 4자만 사용해도 된다. 유일하기만 하면 짧은 SHA-1 값이라도 괜찮다.
먼저 `git log` 명령으로 어떤 커밋이 있는지 조회한다:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800
```

```
    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

`git show` 명령으로 `1c002dd....`로 시작하는 커밋을 조회한다면 아래와 같이 조회 할 수 있다. 다음 명령어는 모두 같다(단 짧은 해시 값이 다른 커밋과 중복되지 않다고 가정):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

`git log` 명령어에 `--abbrev-commit` 옵션을 추가하면 짧은 해시 값을 보여준다. 기본으로 7자를 보여주고 해시 값이 중복되면 더 긴 해시 값을 보여준다:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

보통은 8자에서 10자 내외로도 충분하다. 이 정도로도 중복되지 않는다. 대규모 프로젝트인 리눅스 커널도 커밋을 가리키는데 해시 값 40자 중에서 12자만 사용한다.

6.1.3 SHA-1 해시 값에 대한 단상

Git을 쓰는 사람 중에서 가능성이 낮긴 하지만 언젠가 SHA-1 값이 중복될까 봐 걱정하는 사람도 있다. 정말 그렇게 되면 어떤 일이 벌어질까?

이미 있는 SHA-1 값을 Git 데이터베이스에 커밋하면 새로운 개체라고 해도 이미 커밋한 것으로 간주된다. 그래서 해당 SHA-1 값의 커밋을 Checkout하면 항상 처음에 저장한 커밋만 Checkout된다.

그러나 해시 값이 중복되는 일은 일어나기 어렵다. SHA-1 값의 크기는 20 바이트(160비트)이다. 해시 값이 중복될 확률이 50%가 되는 데 필요한 개체의 수는 2^{80}이다. 이 수는 1.2 자('자'는 '경'의 '억'배 - `10□24`)이다(충돌 확률을 구하는 공식은 `p=(n(n-1)/2) * (1/2□160)`이다). 즉, 지구에 존재하는 모래알의 수에 1200을 곱한 수와 맞먹는다.

아직도 SHA-1 해시 값이 중복될까 봐 걱정하는 사람들을 위해 좀 더 덧붙이겠다. 지구에서 약 6.5억 명의 인구가 개발하고 각자 매초 리눅스 커널 히스토리 전체와(100만 개) 맞먹는 개체를 쏟아 내고 바로 Push한다고 가정하자. 이런 상황에서 해시 값의 충돌 날 확률이 50%가 되기까지는 5년이 걸린다. 그냥 어느 날 동료가 전부 한순간에 늑대에게 물려 죽을 확률이 훨씬 더 높다.

6.1.4 브랜치로 가리키기

브랜치를 사용하는 것이 커밋을 나타내는 가장 쉬운 방법이다. 커밋 개체나 SHA-1 값이 필요한 곳이면 브랜치 이름을 사용할 수 있다. 만약 `topic1` 브랜치의 최근 커밋을 보고 싶으면 아래와 같이 실행한다. `topic1` 브랜치가 `ca82a6d`를 가리키고 있기 때문에 두 명령의 결과는 같다:

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

브랜치가 가리키는 개체의 SHA-1 값에 대한 궁금증은 `rev-parse`이라는 Plumbing 도구가 해결해 준다. 9장에서 이 도구에 대해 좀 더 자세히 설명한다. 기본적으로 `rev-parse`은 저수준 명령어이기 때문에 평소에는 전혀 필요하지 않지만 그래도 한번 사용해보고 어떤 결과가 나오는지 알아 두자:

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

6.1.5 RefLog로 가리키기

Git은 자동으로 브랜치와 HEAD가 지난 몇 달 동안에 가리켰었던 커밋을 모두 기록하는데 이 로그를 Reflog라고 부른다.
`git reflog`를 실행하면 Reflog를 볼 수 있다:

```
$ git reflog
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
1c002dd HEAD@{2}: commit: added some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Git은 브랜치가 가리키는 것이 변경될 때마다 그 정보를 임시 영역에 저장한다. 그래서 예전에 뭘 가리켰었는지 확인할 수 있다. `@{n}` 규칙을 사용하면 아래와 같이 HEAD가 5번 전에 가리켰던 것을 알 수 있다:

```
$ git show HEAD@{5}
```

순서뿐 아니라 시간도 가능하다. 어제 날짜의 `master` 브랜치를 보고 싶으면 아래와 같이 한다:

```
$ git show master@{yesterday}
```

이 명령은 어제 master 브랜치가 가리키고 있던 것이 무엇인지 보여준다. Reflog에 남아있는 것만 조회할 수 있기 때문에 너무 오래된 커밋은 조회할 수 없다.
`git log -g` 명령을 사용하면 `git reflog` 결과를 `git log` 명령과 같은 형태로 볼 수 있다:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

reflog의 일은 모두 로컬의 일이기 때문에 내 reflog가 동료의 저장소에는 있을 수 없다. 이제 막 Clone한 저장소에도 아무것도 한게 없어서 reflog가 하나도 없다. `git show HEAD@{2.months.ago}` 같은 명령은 적어도 두 달 전에 Clone한 저장소에서나 사용할 수 있다. 그러니까 이 명령을 5분 전에 Clone한 저장소에 사용하면 아무것도 나오지 않는다.

6.1.6 계통 관계로 가리키기

계통 관계로도 커밋을 표현할 수 있다. 이름 끝에 ⌧를 붙이면 Git은 해당 커밋의 부모를 찾는다. 프로젝트 히스토리가 아래와 같을 때:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
¦\
¦ * 35cfb2b Some rdoc changes
* ¦ 1c002dd added some blame and merge stuff
¦/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list
```

HEAD^는 바로 "HEAD의 부모"를 의미하므로 바로 이전 커밋을 보여준다:

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'
```

^ 뒤에 숫자도 사용할 수 있다. 예를 들어 d921970^2는 "d921970의 두 번째 부모"를 의미하기에 두 번째 부모가 있는 Merge 커밋에만 사용할 수 있다. 첫 번째 부모는 Merge할 때 Checkout했던 브랜치를 말하고 두 번째 부모는 Merge한 대상 브랜치를 의미한다.

```
$ git show d921970^
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff

$ git show d921970^2
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548
Author: Paul Hedderly <paul+git@mjr.org>
Date:   Wed Dec 10 22:22:03 2008 +0000

    Some rdoc changes
```

계통을 표현하는 방법으로 ~라는 것도 있다. HEAD~와 HEAD^는 똑같이 첫 번째 부모를 가리킨다. 하지만 그 뒤에 숫자를 사용하면 달라진다. HEAD~2는 명령을 실행할 시점의 "첫 번째 부모의 첫 번째 부모", 즉 "조부모"를 가리킨다. 위의 예제에서 HEAD~3은 아래와 같다:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

이 것은 HEAD⊠⊠⊠와 같은 표현이다. 다시 말해서 첫 번째 부모의 첫 번째 부모의 첫 번째 부모를 말한다:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

이 두 표현을 같이 사용할 수도 있다. 위의 예제에서 HEAD~3⊠2를 사용하면 증조부모의 Merge 커밋의 두 번째 부모를 조회한다.

6.1.7 범위로 커밋 가리키기

커밋을 하나씩 조회할 수도 있지만, 범위를 주고 여러 커밋을 한꺼번에 조회할 수도 있다. 범위을 주고 조회하면 브랜치를 관리할 때 유용하다. 브랜치가 상당히 많고 "왜 아직도 주 브랜치에 Merge가 안된 브랜치들은 무엇에 대한 브랜치일까?"라는 의문이 들면 범위를 주고 어떤 브랜치인지 쉽게 알아볼 수 있다.

Double Dot

범위를 표현하는 문법으로 Double Dot(..)을 많이 쓴다. Double Dot은 한쪽에는 있고 다른 쪽에는 없는 커밋이 무엇인지 Git에게 물어보는 것이다. 예들 들어 그림 6-1과 같은 커밋 히스토리가 있다고 가정하자.

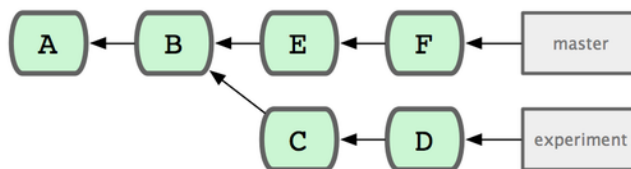



그림 6.1: 범위를 설명하는 데 사용할 예제

experiment 브랜치의 커밋들 중에서 아직 master 브랜치에 Merge하지 않은 것만 보고 싶으면 master..experiment라고 사용한다. 이 표현은 "master에는 없지만, experiment에는 있는 커밋"을 의미한다. 여기에서는 설명을 쉽게 하고자 실제 조회 결과가 아니라 그림 6-1의 문자를 사용한다:

```
$ git log master..experiment
D
C
```

반대로 experiment에는 없고 master에만 있는 커밋이 궁금하면 브랜치 순서를 거꾸로 사용한다. experiment..master는 experiment에는 없고 master에만 있는 것을 알려준다:

```
$ git log experiment..master
F
E
```

`experiment` 브랜치를 Merge하기 전에 무엇이 변경됐는지 궁금하다. 그리고 리모트 저장소에 Push할 때에도 마찬가지로 차이가 궁금하다. 이렇게 궁금한 상황에서 굉장히 유용하다:

```
$ git log origin/master..HEAD
```

이 명령은 `origin` 저장소의 `master` 브랜치에는 없고 현재 Checkout중인 브랜치에만 있는 커밋을 보여준다. Checkout한 브랜치가 `origin/master`라면 `git log origin/master..HEAD`가 보여주는 커밋이 Push하면 서버에 전송될 커밋이다. 그리고 한쪽의 레퍼런스를 생략하면 Git은 HEAD라고 가정한다. `git log origin/master..`는 `git log origin/master..HEAD`와 같다.

세 개 이상의 레퍼런스

Double Dot은 간단하고 유용하다. 하지만, 두 개 이상의 브랜치에는 사용할 수 없다. 그러니까 현재 작업 중인 브랜치에는 있지만 다른 여러 브랜치에는 없는 커밋이 보고 싶으면 `..`으로는 확인할 수 없다. ⌧과 `--not` 옵션 뒤에 브랜치 이름을 넣으면 그 브랜치에 없는 커밋을 찾아준다. 다음 명령어는 모두 같은 명령이다:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Double Dot으로는 세 개 이상의 레퍼런스에 사용할 수 없지만 이 옵션은 가능하다. 예를 들어 `refA`나 `refB`에는 있지만 `refC`에는 없는 커밋을 보려면 다음 중 하나를 사용한다:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

이 조건을 잘 응용하면 작업 중인 브랜치와 다른 브랜치를 매우 상세하게 비교할 수 있다.

Triple Dot

Triple Dot은 양쪽에 있는 두 레퍼런스 사이에서 공통으로 가지는 것을 제외하고 서로 다른 커밋만 보여준다. 그림 6-1의 커밋 히스토리를 다시 보자. 만약 `master`와 `experiment`의 공통부분은 빼고 다른 커밋만 보고 싶으면 아래와 같이 하면 된다:

```
$ git log master...experiment
F
E
D
C
```

우리가 아는 `log` 명령의 결과를 최근 날짜순으로 보여준다. 이 예제에서는 커밋을 네 개 보여준다. 그리고 `log` 명령에 `--left-right` 옵션을 추가하면 각 커밋이 어느 브랜치에 속하는지도 보여주기 때문에 좀 더 이해하기 쉽다:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

위와 같은 명령들을 사용하면 원하는 커밋을 좀 더 꼼꼼하게 살펴볼 수 있다.

6.2 대화형 명령어

Git은 대화형 명령어도 제공해서 좀 더 쉽게 사용할 수 있다. 여기서 소개하는 몇 가지 대화형 명령어를 이용하면 바로 전문가처럼 능숙하게 커밋할 수 있다. 대화형으로 커밋할 파일을 고를 수도 있고 수정된 파일의 일부분만 커밋할 수도 있다. 수정한 파일이 매우 많고 통째로 커밋하지 않고 이슈 별로 나눠서 커밋할 때 유용하다. 이슈 별로 나눠서 커밋하면 동료가 쉽게 검토할 수 있다. `git add` 명령에 `-i`나 `--interactive` 옵션을 주고 실행하면 Git은 아래와 같은 대화형 모드로 들어간다:

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now>
```

이 명령어는 Staging Area의 현재 상태가 어떻고 할 수 있는 일이 무엇인지 보여준다. 기본적으로 `git status` 명령이 보여주는 것과 같지만 좀 더 간결하고 정돈돼 있다. 왼쪽에는 Staged 상태인 파일들을 보여주고 오른쪽에는 Unstaged인 파일들을 보여준다.

그리고 마지막 `Commands` 부분에서는 할 수 일이 무엇인지 보여준다. 파일을 Stage하고 Unstage하는 것, Untracked 상태의 파일을 추가하는 것, Stage한 파일을 diff해보는 것을 할 수 있다. 게다가 수정한 파일의 일부분만 Staging Area에 추가할 수도 있다.

6.2.1 Staging Area에 파일 추가하고 추가 취소하기

`What now>` 프롬프트에서 `2`나 `u`를(update) 입력하면 Staging Area에 추가할 수 있는 파일을 전부 보여준다:

```
What now> 2
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

TODO와 index.html 파일을 Stage하려면 아래와 같이 입력한다:

```
Update>> 1,2
           staged     unstaged path
* 1:    unchanged        +0/-1 TODO
* 2:    unchanged        +1/-1 index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Update>>
```

`*` 표시가 붙은 파일은 stage하도록 선택한 것이다. 선택하고 `Update>>` 프롬프트에 아무것도 입력하지 않고 엔터를 치면 Git은 선택한 파일을 Staging Area로 추가한다:

```
Update>>
updated 2 paths

*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

이제 TODO와 index.html 파일은 Stage했고 simplegit.rb 파일만 아직 Unstaged 상태로 남아 있다. 이제 TODO 파일을 다시 Unstage하고 싶으면 `3`이나 `r`을(Revert) 입력한다:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 3
           staged     unstaged path
  1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> 1
           staged     unstaged path
* 1:        +0/-1      nothing TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

다시 Status를 선택하면 TODO 파일이 Unstaged 상태인 것을 알 수 있다:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:    unchanged        +5/-1 lib/simplegit.rb
```

Staged 파일의 변경내용을 보려면 `6`이나 `d`를(diff) 입력한다. 그러면 먼저 Staged 상태인 파일을 보여준다. 그리고 그중에서 파일 하나를 선택한다. 그 결과는 명령 줄에서 `git diff --cached`라고 실행한 결과와 같다:

```
*** Commands ***
  1: status     2: update      3: revert     4: add untracked
  5: patch      6: diff        7: quit       8: help
What now> 6
           staged     unstaged path
  1:        +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
```

```
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

 <p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

 <script type="text/javascript">
```

대화형 모드를 사용하면 Staging Area에 파일을 좀 더 쉽게 추가할 수 있다.

6.2.2 파일의 일부분만 Staging Area에 추가하기

파일의 일부분만 Staging Area에 추가하는 것도 가능하다. 예를 들어 simplegit.rb 파일은 고친 부분이 두 군데이다. 그 중 하나를 추가하고 나머지는 그대로 두고 싶다. Git에서는 이런 작업도 매우 쉽게 할 수 있다. 대화형 프롬프트에서 `5`, `p`를(patch) 입력한다. 그러면 Git은 부분적으로 Staging Area에 추가할 파일이 있는지 묻는다. 파일을 선택하면 파일의 특정 부분을 Staging Area에 추가할 것인지 부분별로 구분하여 묻는다:

```
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
   end

   def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
   end

   def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,?]?
```

여기에서 `?`를 입력하면 선택 가능한 명령어를 설명해준다:

```
Stage this hunk [y,n,a,d,/,j,J,g,e,?]? ?
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
```

```
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help
```

y나 n을 입력하면 각 부분을 Stage할지 말지 결정할 수 있다. 하지만, 파일을 통째로 stage하거나 필요할 때까지 아예 그대로 남겨 두는 것이 다음에 더 유용할지도 모른다. 어쨌든 파일의 한 부분은 Stage하고 다른 부분은 unstaged 상태로 남겨놓고 status 명령으로 확인해보면 결과는 아래와 같다:

```
What now> 1
           staged     unstaged path
  1:    unchanged        +0/-1 TODO
  2:        +1/-1      nothing index.html
  3:        +1/-1        +4/-0 lib/simplegit.rb
```

simplegit.rb 파일의 상태를 보자. 어떤 줄은 Staged 상태이고 어떤 줄은 Unstaged라고 알려줄 것이다. 이 파일은 부분적으로 Stage하였다. 이제 대화형 모드를 종료하고 일부분만 Stage한 파일을 커밋할 수 있다.

끝으로 대화형 스크립트로만 파일 일부분을 Stage할 수 있는 것은 아니다. `git add -p`나 `git add --patch`로도 같은 일을 할 수 있다.

6.3 Stashing

당신이 어떤 프로젝트에서 한 부분을 담당하고 있다고 하자. 그리고 여기에서 뭔가 작업하던 일이 있고 다른 요청이 들어와서 잠시 브랜치를 변경해야 할 일이 생겼다고 치자. 아직 완료하지 않은 일을 커밋하는 것은 좀 껄끄럽다. 이런 상황에서는 커밋하지 않고 나중에 다시 돌아와서 작업을 다시 하고 싶을 것이다. 이 문제는 `git stash`라는 명령으로 해결할 수 있다.

Stash 명령을 사용하면 워킹 디렉토리에서 수정한 파일만 저장한다. Stash는 Modified이면서 Tracked 상태인 파일과 Staging Area에 있는 파일들을 보관해두는 장소다. 아직 끝나지 않은 수정사항을 스택에 잠시 저장했다가 나중에 다시 적용할 수 있다.

6.3.1 하던 일을 Stash하기

예제 프로젝트를 하나 살펴보자. 파일을 두 개 수정하고 그 중 하나는 Staging Area에 추가한다. 그리고 `git status` 명령을 실행하면 아래와 같은 결과를 볼 수 있다:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

이제 브랜치를 변경한다. 아직 작업 중인 파일은 커밋할 게 아니라서 모두 Stash한다. `git stash`를 실행하면 스택에 새로운 Stash가 만들어진다:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

대신 워킹 디렉토리는 깨끗해졌다:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

이제 아무 브랜치나 골라서 바꿀 수 있다. 수정하던 것은 스택에 저장했다. 아래와 같이 `git stash list`를 사용하여 저장한 Stash를 확인한다:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

Stash 두 개는 원래 있었던 것이다. 그래서 현재 총 세 개의 Stash를 사용할 수 있다. 이제 `git stash apply`를 사용하여 Stash를 적용할 수 있다. `git stash` 명령을 실행하면 이 명령에 대한 도움말을 보여주기 때문에 편리하다. 다른 Stash를 고르고 싶으면 Stash 이름을 입력해야 한다. 이름이 없으면 Git은 가장 최근의 Stash를 적용한다:

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   index.html
#      modified:   lib/simplegit.rb
#
```

Git은 Stash에 저장할 때 수정하던 파일을 복원해준다. 복원할 때의 워킹 디렉토리는 Stash할 때의 그 브랜치이고 워킹 디렉토리도 깨끗한 상태였다. 하지만, 꼭 깨끗한 워킹 디렉토리나 Stash할 때와 같은 브랜치에 적용해야 하는 것은 아니다. 어떤 브랜치에서 Stash하고 다른 브랜치로 옮기고서 거기에 Stash를 복원할 수 있다. 그리고 꼭 워킹 디렉토리가 깨끗한 상태일 필요도 없다. 워킹 디렉토리에 수정하고 커밋하지 않은 파일들이 있을 때에도 Stash를 적용할 수 있다. 만약 충돌이 나면 알려준다.

Git은 Stash를 적용할 때 Staged 상태였던 파일을 자동으로 다시 Staged 상태로 만들어 주지 않는다. 그래서 `git stash apply` 명령을 실행할 때 `--index` 옵션을 주어야 Staged 상태까지 복원한다. 그럼 원래 작업하던 상태로 돌아올 수 있다:

```
$ git stash apply --index
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#      modified:   lib/simplegit.rb
#
```

`apply` 옵션은 단순히 Stash를 적용하는 것뿐이다. Stash는 여전히 스택에 남아 있다. `git stash drop` 명령을 사용하여 해당 Stash를 제거한다:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

그리고 `git stash pop`이라는 명령도 있는데 이 명령은 Stash를 적용하고 나서 바로 스택에서 제거해준다.

6.3.2 Stash 되돌리기

Stash를 적용하고 나서 아차 싶을 때에는 다시 되돌려 놓아야 한다. Git은 `stash unapply` 같은 명령을 제공하지는 않는다. 하지만, Stash를 이용해서 패치를 만들고 그것을 거꾸로 적용할 수 있다:

```
$ git stash show -p stash@{0} | git apply -R
```

Stash를 명시하지 않으면 Git은 가장 최근의 Stash를 사용한다:

```
$ git stash show -p | git apply -R
```

`stash-unapply`라는 alias를 만들고 편리하게 할 수도 있다:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash apply
$ #... work work work
$ git stash-unapply
```

6.3.3 Stash를 적용한 브랜치 만들기

보통 Stash에 저장하면 한동안 그대로 유지하고 그 브랜치에서는 계속 새로운 일을 한다. 그러면 저장한 Stash를 적용하는 것이 문제가 될 수 있다. 수정한 파일에 Stash를 적용하면 충돌이 날 수 있다. 충돌이 나면 충돌을 해결해야 한다. 그리고 Stash한 것은 다시 테스트해야 한다. `git stash branch` 명령을 실행하면 Stash할 당시의 커밋을 Checkout한 후 새로운 브랜치를 만들고 여기에 적용한다. 이 모든 것이 성공하면 Stash를 삭제한다:

```
$ git stash branch testchanges
Switched to a new branch "testchanges"
# On branch testchanges
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
```

```
#      modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

이 명령은 브랜치를 새로 만들고 Stash를 복원해주는 매우 편리한 도구다.

6.4 히스토리 단장하기

Git으로 일하다 보면 어떤 이유로든 커밋 히스토리를 수정해야 할 때가 있다. 결정을 나중으로 미룰 수 있던 것은 Git의 장점이다. Staging Area가 있어서 커밋할 파일을 고르는 일을 커밋하는 순간으로 미룰 수 있고 Stash 명령으로 하던 일을 미룰 수 있다. 게다가 이미 커밋한 내용을 수정할 수 있다. 거의 모든 것을 수정할 수 있다. 커밋 순서도 변경할 수 있고 커밋 메시지와 커밋한 파일도 변경할 수 있다. 여러 개의 커밋을 하나로 합치거나 반대로 하나의 커밋을 여러 개로 분리할 수도 있다. 아니면 커밋 전체를 삭제할 수도 있다. 하지만, 이 모든 것은 다른 사람과 코드를 공유하기 전에 해야 한다.
이 절에서는 사람들과 코드를 공유하기 전에 커밋 히스토리를 예쁘게 단장하는 방법에 대해서 설명한다.

6.4.1 마지막 커밋을 수정하기

히스토리를 단장하는 일 중에서는 마지막 커밋을 수정하는 것이 가장 자주 하는 일이다. 기본적으로 두 가지로 나눌 수 있는데 하나는 커밋 메시지를 수정하는 것이고 다른 하나는 파일 목록을 수정하는 것이다.
커밋 메시지를 수정하는 방법은 매우 간단하다:

```
$ git commit --amend
```

이 명령은 자동으로 텍스트 편집기를 실행시켜서 마지막 커밋 메시지를 열어준다. 여기에 메시지를 수정하고 편집기를 닫으면 편집기는 수정한 메시지로 마지막 커밋을 수정한다.
커밋하고 나서 새로 만들었거나 다시 수정한 파일을 마지막 커밋에 포함할 수 있다. 기본적으로 방법은 같다. 파일을 수정하고 `git add` 명령으로 Staging Area에 넣거나 `git rm` 명령으로 파일 삭제한다. 그리고 `git commit --amend` 명령으로 커밋하면 된다. 이 명령은 현 Staging Area의 내용을 이용해서 수정한다.
이때 SHA-1 값이 바뀌기 때문에 과거의 커밋을 변경할 때 주의해야 한다. rebase처럼 이미 Push한 커밋은 수정하면 안 된다.

6.4.2 커밋 메시지를 여러 개 수정하기

최근 커밋이 아니라 예전 커밋을 수정하려면 다른 도구가 필요하다. 히스토리 수정용 도구는 없지만 `rebase` 명령을 이용하여 수정할 수 있다. 현재 작업하는 브랜치에서 각 커밋을 하나하나 수정하는 것이 아니라 어느 시점부터 HEAD까지의 커밋을 한 번에 Rebase한다. 대화형 Rebase 도구를 사용하면 커밋을 처리할 때마다 멈춘다. 그러면 각 커밋의 메시지를 수정하거나 파일을 추가하고 변경하는 등의 일을 진행할 수 있다. `git rebase` 명령에 `-i` 옵션을 추가하면 대화형 모드로 Rebase할 수 있다. 어떤 시점부터 HEAD까지 Rebase할 것인지 아규먼트로 넘기면 된다.

마지막 커밋 메시지 세 개를 모두 수정하거나 그 중 몇 개를 수정하는 시나리오를 살펴보자. `git rebase -i`의 아규먼트로 편집하려는 마지막 커밋의 부모를 `HEAD~2`나 `HEAD~3`로 해서 넘긴다. 마지막 세 개의 커밋을 수정하는 것이기 때문에 `~3`이 좀 더 기억하기 쉽다. 그렇지만, 실질적으로 가리키게 되는 것은 수정하려는 커밋의 부모인 네 번째 이전 커밋이다.

```
$ git rebase -i HEAD~3
```

이 명령은 rebase하는 것이기 때문에 메시지의 수정 여부에 관계없이 `HEAD~3..HEAD` 범위에 있는 모든 커밋을 수정한다. 다시 강조하지만 이미 중앙서버에 Push한 커밋은 절대 고치지 말아야 한다. Push한 커밋을 Rebase하면 결국 같은 내용을 두 번 Push하는 것이기 때문에 다른 개발자들이 혼란스러워 한다.

실행하면 텍스트 편집기가 열리고 그 안에는 수정하려는 커밋 목록이 첨부된다:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

이 커밋은 모두 `log` 명령과는 정반대의 순서로 나열된다. `log` 명령을 실행하면 아래와 같은 결과를 볼 수 있다:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

위 결과의 역순임을 기억하자. 대화형 rebase는 스크립트에 적혀 있는 순서대로 `HEAD~3`부터 적용하기 시작하고 위에서 아래로 각각의 커밋을 순서대로 수정한다. 순서대로 적용하는 것이기 때문에 제일 위에 있는 것이 최신이 아니라 가장 오래된 것이다.

특정 커밋에서 실행을 멈추게 하려면 스크립트를 수정해야 한다. `pick`이라는 단어를 `edit`로 수정하면 그 커밋에서 멈춘다. 가장 오래된 커밋 메시지를 수정하려면 아래와 같이 편집한다:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

저장하고 편집기를 종료하면 Git은 목록에 있는 커밋 중에서 가장 오래된 커밋으로 이동하고, 아래와 같은 메시지를 보여주고, 명령 프롬프트를 보여준다:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with

       git commit --amend

Once you're satisfied with your changes, run

       git rebase --continue
```

정확히 뭘 해야 하는지 알려준다. 아래와 같은 명령을 실행하고:

```
$ git commit --amend
```

커밋 메시지를 수정하고 텍스트 편집기를 종료한다. 그리고 아래 명령어를 실행한다:

```
$ git rebase --continue
```

이렇게 나머지 두 개의 커밋에 적용하면 끝이다. 다른 것도 `pick`을 `edit`로 수정해서 이 작업을 몇 번이든 반복할 수 있다. Git이 멈출 때마다 커밋을 수정할 수 있고 완료할 때까지 계속 할 수 있다.

6.4.3 커밋 순서 바꾸기

대화형 Rebase 도구로 커밋 전체를 삭제하고 순서도 바꿀 수 있다. "added cat-file" 커밋을 삭제하고 다른 두 커밋의 순서를 변경하려면 이 rebase 스크립트를:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

아래와 같이 수정한다:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

수정한 내용을 저장하고 편집기를 종료하면 Git은 브랜치를 이 커밋들의 부모로 이동시키고서 `310154e`와 `f7f3f6d`를 순서대로 적용한다. 그러면 커밋 순서가 변경됐고 “added cat-file” 커밋이 제거된 것을 확인할 수 있다.

6.4.4 커밋 합치기

대화형 Rebase 명령을 이용하여 여러 개의 커밋을 꾹꾹 눌러서 하나의 커밋으로 만들어 버릴 수 있다. Rebase 스크립트에 자동으로 포함된 도움말에 설명돼 있다:

```
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

“pick”이나 “edit”말고 “squash”를 입력하면 Git은 해당 커밋과 바로 이전 커밋을 합칠 것이고 커밋 메시지도 Merge한다. 그래서 3개의 커밋을 모두 합치려면 스크립트를 아래와 같이 수정한다:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

저장하고 나서 편집기를 종료하면 Git은 3개의 커밋 메시지를 Merge할 수 있도록 에디터를 바로 실행해준다:

```
# This is a combination of 3 commits.
# The first commit's message is:
changed my name a bit

# This is the 2nd commit message:

updated README formatting and added blame

# This is the 3rd commit message:

added cat-file
```

이 메시지를 저장하면 3개의 커밋이 모두 합쳐진 하나의 커밋만 남는다.

6.4.5 커밋 분리하기

커밋을 분리한다는 것은 기존 커밋을 Reset하고(혹은 되돌려 놓고) Stage를 여러 개로 분리하고 나서 그것을 원하는 횟수만큼 다시 커밋하는 것이다. 예로 들었던 커밋 세 개 중에서 가운데 것을 분리해보자. 이 커밋은 "updated README formatting and added blame"라는 커밋인데 "updated README formatting"과 "added blame"으로 분리해보자. `rebase -i` 스크립트에서 해당 커밋을 "edit"로 변경한다:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

위와 같이 수정하고 나서 저장하고 편집기를 종료하면 Git은 제일 오래된 커밋의 부모로 이동하고서 `f7f3f6d`과 `310154e`을 처리하고 콘솔 프롬프트를 보여준다. 여기서 커밋을 해제하는 `git reset HEAD^`라는 명령으로 커밋을 해제한다. 그러면 수정했던 파일은 Unstaged 상태가 된다. 그다음에 파일들을 Stage한 후 커밋하는 일을 원하는 만큼 반복하고 나서 `git rebase --continue`라는 명령을 실행하면 남은 Rebase작업이 끝난다:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
```

```
$ git commit -m 'added blame'
$ git rebase --continue
```

나머지 `a5f4a0d` 커밋도 처리되면 히스토리는 아래와 같다:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

다시 강조하지만, 목록에 있는 모든 커밋의 SHA-1 값은 변경된다. 그래서 이미 서버에 Push한 커밋을 수정하면 안된다.

6.4.6 filter-branch는 포크레인

수정해야 하는 커밋이 너무 많아서 rebase 스크립트로 수정하기 어려울 것 같으면 다른 방법을 사용하는 것이 좋다. 모든 커밋의 이메일 주소를 변경하거나 어떤 파일을 삭제하는 경우를 살펴보자. `filter-branch`라는 명령으로 수정할 수 있는데 rebase가 삽이라면 이 명령은 포크레인이라고 할 수 있다. `filter-branch`도 역시 수정하려는 커밋이 이미 공개돼서 다른 사람과 함께 공유하는 중이라면 사용하지 말아야 한다. 하지만, 잘 쓰면 꽤 유용하다. `filter-branch`가 어떤 경우에 유용할지 예를 들어서 설명한다.

모든 커밋에서 파일을 제거하기

갑자기 누군가 생각 없이 `git add .` 같은 명령어를 실행해 버려서 공룡 똥 덩어리가 커밋됐거나 실수로 암호가 포함된 파일을 커밋해서 이런 파일들을 다시 삭제해야 하는 상황을 살펴보자. 이런 상황은 생각보다 자주 발생한다. `filter-branch`는 히스토리 전체에서 필요한 것만 골라내는 데 사용하는 도구다. `filter-branch`의 `--tree-filter`라는 옵션을 사용하면 히스토리에서 passwords.txt라는 파일을 아예 제거할 수 있다:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

`--tree-filter` 옵션은 프로젝트를 Checkout한 후에 각 커밋에 명시한 명령어를 실행시키고 그 결과를 다시 커밋한다. 이 예제에서는 각 스냅샷에 passwords.txt라는 파일이 있으면 그 파일을 삭제한다. 실수로 편집기의 백업파일을 커밋했으면 `git filter-branch --tree-filter "find * -type f -name '*~' -delete" HEAD`라고 실행해서 삭제할 수 있다.

이 명령은 모든 파일과 커밋을 정리하고 브랜치 포인터를 다시 복원해준다. 테스팅 브랜치에서 사용할 명령을 점검하고 나서 master 브랜치를 정리한다. 그리고 `filter-branch` 명령에 `--all` 옵션을 추가하면 모든 브랜치에 적용된다.

하위 디렉토리를 루트 디렉토리로 만들기

다른 VCS에서 코드를 임포트하면 그 VCS만을 위한 디렉토리가 있을 수 있다. SVN에서 코드를 임포트하면 trunk, tags, branch 디렉토리가 포함된다. 모든 커밋에 대해 `trunk` 디렉토리를 프로젝트 루트 디렉토리로 만들 때에도 `filter-branch` 명령이 유용하다:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

이제 `trunk` 디렉토리를 루트 디렉토리로 만들었다. Git은 입력한 디렉토리와 관련이 없는 커밋을 자동으로 삭제한다.

모든 커밋의 이메일 주소를 수정하기

프로젝트를 오픈소스로 공개할 때에도 회사 이메일 주소로 커밋된 것을 개인 이메일 주소로 변경해야 한다. 아니면 아예 `git config`로 이름과 이메일 주소를 설정하는 것을 잊었을 수도 있다. 어쨌든 `filter-branch` 명령의 `--commit-filter` 옵션을 사용하여 각 커밋에 등록된 이메일 주소를 수정할 수 있다. 이메일 주소를 변경할 때는 조심해야 한다.

```
$ git filter-branch --commit-filter '
        if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
        then
                GIT_AUTHOR_NAME="Scott Chacon";
                GIT_AUTHOR_EMAIL="schacon@example.com";
                git commit-tree "$@";
        else
                git commit-tree "$@";
        fi' HEAD
```

이메일 주소를 새 주소로 변경했다. 모든 커밋은 부모의 SHA-1 값을 가지고 있기 때문에 조건에 만족하는 커밋의 SHA-1값만 바뀌는 것이 아니라 모든 커밋의 SHA-1 값이 바뀐다.

6.5 Git으로 버그 찾기

Git은 굉장히 유연해서 어떤 프로젝트에나 사용할 수 있다. 게다가 문제를 일으킨 범인이나 버그도 쉽게 찾을 수 있도록 도와준다.

6.5.1 파일 어노테이션

버그를 찾을 때 먼저 그 코드가 왜, 언제 추가했는지 알고 싶을 것이다. 이때는 파일 어노테이션을 활용한다. 한줄한줄 마지막으로 커밋한 사람이 누구인지, 언제 마지막으로 커밋했는지 볼 수 있다. 어떤 메소드에 버그가 있으면 `git blame` 명령으로 그 메소드의 각 줄을 누가 언제 마지막으로 고쳤는지 찾아낼 수 있다:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 12)  def show(tree = 'master')
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 14)  end
^4832fe2 (Scott Chacon  2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 16)  def log(tree = 'master')
79eaf55d (Scott Chacon  2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 18)  end
9f6560e4 (Scott Chacon  2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20)  def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22)  end
```

첫 항목은 그 줄을 마지막에 수정한 커밋의 SHA-1 값이다. 그다음 두 항목은 누가, 언제 그 줄을 커밋했는지 보여준다. 그래서 누가, 언제 커밋했는지 쉽게 찾을 수 있다. 그 뒤에 파일의 줄 번호와 내용을 보여준다. 그리고 ▯4832fe2 커밋이 궁금할 텐데 이 표시가 붙어 있으면 해당 줄이 처음 커밋한 것을 의미한다. 그러니까 해당 줄은 4832fe2에서 커밋된 후 변경된 적이 없다. 지금까지 커밋을 수정하는 것을 배우면서 ▯을 적어도 세 곳에서 사용한다고 배웠기 때문에 약간 헷갈릴 수 있으니 혼동하지 말자.

Git은 파일 이름을 변경한 이력을 별도로 기록해두지 않는다. 하지만, 원래 이 정보들은 각 스냅샷에 저장되고 이 정보를 이용하여 변경 이력을 만들어 낼 수 있다. 그러니까 파일에 생긴 변화는 무엇이든지 알아낼 수 있다. Git은 파일 어노테이션을 분석하여 코드들이 원래 어떤 파일에서 커밋된 것인지 찾아준다. 예를 들어보자. GITServerHandler.m을 여러 개의 파일로 리팩토링했는데 그 중 한 파일이 GITPackUpload.m이라는 파일이라고 하자. -C 옵션으로 GITPackUpload.m 파일을 추적해보면 각 코드가 원래 어떤 파일로 커밋된 것인지 알 수 있다:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)        //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 146)        NSString *parentSha;
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 147)        GITCommit *commit = [g
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 149)        //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m    (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)        if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)                [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

언제나 코드가 커밋될 당시의 파일이름을 알 수 있기 때문에 코드를 어떻게 리팩토링해도 추적할 수 있다. 그리고 어떤 파일에 적용해봐도 각 줄을 커밋할 당시의 파일이름을 알 수 있다. 버그를 찾을 때 정말 유용하다.

6.5.2 이진 탐색

파일 어노테이션은 특정 이슈와 관련된 커밋을 찾는 데에도 좋다. 문제가 생겼을 때 의심스러운 커밋이 수십, 수백 개에 이르면 도대체 어디서부터 시작해야 할지 모를 수 있다. 이때는 `git bisect` 명령이 유용하다. `bisect` 명령은 커밋 히스토리를 이진 탐색 방법으로 좁혀 주기 때문에 이슈와 관련된 커밋을 최대한 빠르게 찾아낼 수 있도록 도와준다.

코드를 운용 환경에 배포하고 난 후에 개발할 때 발견하지 못한 버그가 있다고 보고받았다. 그런데 왜 그런 현상이 발생하는지 아직 이해하지 못하는 상황을 가정해보자. 해당 이슈를 다시 만들고 작업하기 시작했는데 뭐가 잘못됐는지 알아낼 수 없다. 이럴 때 bisect를 사용하여 코드를 뒤져 보는 게 좋다. 먼저 `git bisect start` 명령으로 이진 탐색을 시작하고 `git bisect bad`를 실행하여 현재 커밋에 문제가 있다고 표시를 남기고 나서 문제가 없는 마지막 커밋을 `git bisect good [good_commit]` 명령으로 표시한다.

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

이 예제에서 마지막으로 괜찮았던 커밋(v1.0)과 현재 문제가 있는 커밋 사이에 있는 커밋은 전부 12개이고 Git은 그 중간에 있는 커밋을 Checkout해준다. 여기에서 해당 이슈가 구현됐는지 테스트해보고 만약 이슈가 있으면 그 중간 커밋 이전으로 범위를 좁히고 이슈가 없으면 그 중간 커밋 이후로 범위를 좁힌다. 이슈를 발견하지 못했으면 `git bisect good`으로 이슈가 아직 없음을 알리고 계속 진행한다:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

현재 문제가 있는 커밋과 지금 테스트한 커밋 사이에서 중간에 있는 커밋이 Checkout됐다. 다시 테스트해보고 이슈가 있으면 `git bisect bad`로 이슈가 있다고 알린다:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

이제 이슈를 처음 구현한 커밋을 찾았다. 이 SHA-1 값을 포함한 이 커밋의 정보를 확인하고 수정된 파일이 무엇인지 확인한다. 이 문제가 발생한 시점에 도대체 무슨 일이 있었는지 아래와 같이 살펴본다:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
```

```
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

    secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcfc639b1a3814550e62d60b8e68a8e4 M  config
```

이제 찾았으니까 `git bisect reset` 명령을 실행시켜서 이진 탐색을 시작하기 전으로 HEAD를 돌려놓는다:

```
$ git bisect reset
```

수백 개의 커밋들 중에서 버그가 만들어진 커밋을 찾는 데 몇 분밖에 걸리지 않는다. 프로젝트가 정상적으로 수행되면 0을 반환하고 문제가 있을 경우 1을 반환하는 스크립트를 만들면 이 `git bisect` 과정을 완전히 자동화 할 수 있다. 먼저 `bisect start` 명령으로 bisect를 사용할 범위를 알려준다. 위에서 한 것처럼 문제가 있다고 아는 커밋과 문제가 없다고 아는 커밋을 넘기면 된다:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

문제가 생긴 첫 커밋을 찾을 때까지 Checkout할 때마다 `test-error.sh`를 실행한다. `make`든지 `make tests`든지 어쨌든 이슈를 찾는 테스트를 실행하여 찾는다.

6.6 서브모듈

프로젝트를 수행하다 보면 다른 프로젝트를 사용해야 하는 경우가 종종 있다. 보통 사용할 프로젝트들은 독립적으로 개발된 라이브러리들이다. 이런 상황에서 자주 생기는 이슈는, 두 프로젝트를 서로 별개로 다루면서도 그 중 하나를 다른 하나 안에서 사용할 수 있어야 한다는 것이다.

Atom 피드를 제공하는 웹사이트를 만든다고 가정하자. Atom 피드를 생성하는 코드는 직접 작성하지 않고 라이브러리를 가져다 쓰기로 했다. 그러면 CPAN이나 Ruby gem 같은 라이브러리 관리 도구를 사용하거나 해당 소스코드를 프로젝트로 복사해야 한다. 사실 라이브러리를 수정하는 것은 어렵다. 하지만 수정한 라이브러리를 모든 사용자가 이용할 수 있도록 배포하는 것은 더 어렵다. 그래서 프로젝트에 라이브러리 코드를 포함시켜서 수정하는 방법도 사용한다. 이렇게 라이브러리 코드를 포함시키면 원래 라이브러리 프로젝트의 코드와 Merge하기 어렵게 된다.

Git의 서브모듈은 이런 문제를 해결해준다. 서브모듈은 Git 저장소 안에 다른 Git 저장소를 둘 수 있게 해준다. 이렇게 해도 두 Git 저장소 모두 여전히 독립적으로 관리된다.

6.6.1 서브모듈 시작하기

한 번 Ruby 웹서버 게이트웨이 인터페이스인 Rack 라이브러리를 프로젝트에 추가해보자. 추가하고 나서도 앞으로 여전히 해당 저장소에서 관리할 수 있기 때문에 마음 놓고 코드를 수정할 수 있다. 먼저 `git submodule add` 명령으로 프로젝트를 서브모듈로 추가한다:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

이제 프로젝트 디렉토리를 보면 `rack`이라는 디렉토리가 생겼을 것이다. 그 디렉토리가 Rack 프로젝트이다. `rack` 디렉토리 안에서 수정하고 Push할 권한이 있는 저장소를 하나 추가하고 나서 그 저장소에 Push한다. 물론 원래 프로젝트 저장소에서도 Fetch하고 Merge할 수 있다. 서브모듈을 추가한 직후 바로 `git status`라는 명령을 실행하면 아래와 같이 두 파일이 생긴 것을 알 수 있다:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#      new file:   .gitmodules
#      new file:   rack
#
```

`.gitmodules` 파일을 살펴보자. 이 것은 로컬 디렉토리와 프로젝트 URL의 매핑 정보가 저장된 설정파일이다:

```
$ cat .gitmodules
[submodule "rack"]
      path = rack
      url = git://github.com/chneukirchen/rack.git
```

서브모듈 개수만큼 이 항목이 생긴다. 이 파일도 `.gitignore` 파일처럼 버전 관리된다. 다른 파일처럼 Push하고 풀한다. 이 프로젝트를 Clone하는 사람은 `.gitmodules` 파일을 보고 어떤 서브모듈 프로젝트가 있는지 알 수 있다.
`.gitmodules`은 살펴봤고 이제 `rack` 항목에 대해 살펴보자. `git diff` 명령을 실행시키면 흥미로운 점을 발견할 수 있다:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 0000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

Git은 `rack` 디렉토리를 서브모듈로 취급하기 때문에 파일들을 직접 추적하지 않고 커밋 하나만 저장한다. `rack` 디렉토리에서 수정을 하고 커밋하면 다른 사람이 같은 환경을 만들 수 있도록 HEAD가 가리키는 커밋이 슈퍼프로젝트에 저장된다.

`master`처럼 브랜치 이름 같은 레퍼런스가 저장되는 것이 아니라 커밋의 SHA-1 값이 저장된다.

슈퍼프로젝트도 커밋해야 된다:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
```

rack 디렉토리의 모드는 160000이다. 160000 모드는 일반적인 파일이나 디렉토리가 아니라는 의미다.

하위 프로젝트의 마지막 커밋이 바뀔 때마다 슈퍼프로젝트에 저장된 커밋도 바꿔준다. `rack` 디렉토리를 별도의 프로젝트로 취급하기 때문에 모든 Git 명령은 독립적으로 동작한다:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack
$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date:   Wed Mar 25 14:49:04 2009 +0100

    Document version change
```

6.6.2 서브모듈이 있는 프로젝트 Clone하기

서브모듈을 사용하는 프로젝트를 Clone하면 해당 서브모듈 디렉토리는 빈 디렉터리다:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
total 8
-rw-r-r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
$
```

분명히 `rack` 디렉토리가 있지만 비워져 있다. 먼저 `git submodule init` 명령으로 서브모듈을 초기화하고 `git submodule update` 명령으로 서버에서 데이터를 가져온다. 데이터를 전부 가져오면 슈퍼프로젝트에 저장된 커밋으로 Checkout된다:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbeb7821e37fa53e69afcf433'
```

`rack` 디렉토리는 이제 복원했다. 그리고 누군가 rack을 수정하면 그 코드를 가져다 Merge한다:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
```

```
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#      modified:   rack
#
```

Merge해서 서브모듈의 HEAD 값이 변경됐다. 슈퍼프로젝트가 아는 커밋과 서브모듈의 HEAD가 달라서 아직 워킹 디렉토리의 상태는 깨끗한 상태가 아니다:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbeb7821e37fa53e69afcf433
```

이럴 때 `git submodule update` 명령을 실행해서 해결한다:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
   08d709f..6c5e70b  master     -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

서브모듈 프로젝트를 풀할 때마다 `git submodule update` 명령을 실행해야 한다. 뭔가 속는 것 같지만 잘 된다.
개발자들이 흔히 저지르는 실수로 서브모듈의 코드를 수정하고 나서 서버에 Push하지 않는 경우가 있다. 슈퍼프로젝트는 Push했지만 프로젝트가 아는 커밋은 아직 Push하지 않고 개발자 PC에만 있다. 만약 다른 개발자가 `git submodule update`를 실행하면 슈퍼프로젝트에 저장된 커밋을 서브모듈 프로젝트에서 찾을 수 없어서 에러가 발생한다:

```
$ git submodule update
fatal: reference isn’t a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'rack'
```

누가 마지막으로 서브모듈을 수정했는지 확인하고:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Apr 9 09:19:14 2009 -0700

    added a submodule reference I will never make public. hahahahaha!
```

그 개발자에게 이메일을 보내거나 전화를 건다.

6.6.3 슈퍼프로젝트

프로젝트 규모가 크면 CVS나 Subversion에서는 모듈 프로젝트를 간단히 하위 디렉토리로 만들었다. 가끔 Git에서도 이런 Workflow을 사용하려는 개발자들이 있다.
Git에서는 각 하위 디렉토리를 별도의 Git 저장소로 만들어야 한다. 그리고 그 저장소를 포함하는 상위 저장소를 만든다. 슈퍼프로젝트의 태그와 브랜치를 이용해서 각 프로젝트의 관계를 구체적으로 정의할 수 있다는 것은 Git만의 장점이다.

6.6.4 서브모듈 사용할 때 주의할 점들

전체적으로 서브모듈은 어렵지 않게 사용할 수 있지만, 서브모듈의 코드를 수정하는 경우에는 주의가 필요하다. `git submodule update` 명령을 실행시키면 특정 브랜치가 아니라 슈퍼프로젝트에 저장된 커밋을 Checkout해 버린다. 그러면 `detached HEAD`라고 부르는 상태가 된다. `detached HEAD`는 HEAD가 브랜치나 태그 같은 간접 레퍼런스를 가리키지 않고 커밋을 가리키는 것을 말한다. 데이터를 잃어 버릴 수도 있기 때문에 일반적으로 `detached HEAD` 상태는 피해야 한다.
`submodule update`를 실행하고 나서 별도의 작업용 브랜치를 만들지 않고 서브모듈 코드를 수정하고 커밋한다. 그리고 나중에 커밋한 것을 잊은 채로 슈퍼프로젝트에서 다시 `git submodule update`를 실행시키면 Git은 아무 말 없이 Checkout해 버린다. 엄밀히 말해서 커밋을 없어진 것은 아니지만 브랜치에 속하지 않는 커밋을 찾아내기란 정말 어렵다.
`git checkout -b work` 같은 명령으로 작업할 때마다 work 브랜치를 만들면 이 문제를 피할 수 있다. 실수로 `submodule update` 명령을 실행해 버려서 하던 일을 놓쳐버려도 포인터가 있어서 언제든지 되찾을 수 있다.
그리고 서브모듈이 있는 슈퍼프로젝트의 브랜치를 오갈 때는 약간의 추가작업이 필요하다. 브랜치를 만들고 서브모듈을 추가한다. 그 다음에 서브모듈이 없는 브랜치로 돌아간다. 그렇지만, 이미 추가한 서브모듈 디렉토리가 untracked 상태로 보인다:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
```

```
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#      rack/
```

서브모듈 디렉토리를 다른 곳에 옮겨 두거나 삭제해야 한다. 삭제할 경우는 원래 브랜치로 돌아왔을 때 서브모듈을 다시 Clone해야 하고, 이 경우 아직 Push하지 않았던 변경사항이나 브랜치를 잃을 수 있다.

rack이라는 디렉토리가 있고 이것을 서브모듈로 바꾸려고 한다고 가정하자. 먼저 rack 디렉토리를 삭제하고 `submodule add`를 실행하면 Git은 아래와 같은 에러를 뱉는다:

```
$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index
```

`rack` 디렉토리를 Staging Area에서 제거하면 서브모듈을 추가할 수 있다.

```
$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
```

한 브랜치에서는 해결했다. 아직 해당 디렉토리를 서브모듈로 만들지 않은 브랜치를 Checkout하려고 하면 아래와 같은 에러가 난다:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

다른 브랜치로 바꾸기 전에 `rack` 서브모듈 디렉토리를 다른 곳으로 옮겨 둔다:

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README rack
```

그리고 나서 다시 서브모듈이 있는 브랜치로 돌아가면 rack 디렉토리는 텅 비어 있다. `git submodule update` 명령으로 다시 Clone하거나 `/tmp/rack/`에 복사해둔 파일을 다시 복사한다.

6.7 Subtree Merge

서브모듈 시스템이 무엇이고 어디에 쓰는지 배웠다. 그런데 같은 문제를 해결하는 방법이 또 하나 있다. Git은 Merge하는 시점에 무엇을 Merge할지, 어떤 전략을 사용할지 결정해야 한다. Git은 브랜치 두 개를 Merge할 때에는 Recursive 전략을 사용하고 세 개 이상의 브랜치를 Merge할 때에는 Octopus 전략을 사용한다. 이 전략은 자동으로 선택된다. Merge할 브랜치가 두개면 Recursive 전략이 선택된다. Recursive 전략은 Merge하려는 두 커밋과 공통 조상 커밋을 이용하는 `three-way merge`를 사용하기 때문에 단 두 개의 브랜치에만 적용할 수 있다. Octopus 전략은 브랜치가 여러 개라도 Merge할 수 있지만 비교적 충돌이 쉽게 일어난다.

다른 전략도 있는데 그중 하나가 Subtree Merge다. 이 Merge는 하위 프로젝트 문제를 해결하는 데에도 사용한다. 위에서 사용했던 Rack 예제를 적용해 보자.

Subtree Merge는 마치 하위 프로젝트가 아예 합쳐진 것처럼 보일 정도로 한 프로젝트를 다른 프로젝트의 하위 디렉토리에 연결해준다. 정말 놀라운 기능이다.

Rack 프로젝트를 리모트 저장소로 추가시키고 브랜치를 Checkout한다:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From git@github.com:schacon/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Checkout한 `rack_branch`의 루트 디렉토리와 origin 프로젝트의 `master` 브랜치의 루트 디렉토리는 다르다. 브랜치를 바꿔가며 어떻게 다른지 확인한다:

```
$ ls
AUTHORS        KNOWN-ISSUES   Rakefile      contrib       lib
COPYING        README         bin           example       test
$ git checkout master
Switched to branch "master"
$ ls
README
```

여기에서 Rack 프로젝트를 `master` 브랜치의 하위 디렉토리에 넣으려면 `git read-tree` 명령어를 사용한다. 9장에서 `read-read` 류의 명령어를 좀 더 자세히 다룬다. 여기에서는 워킹 디렉토리와 Staging Area로 어떤 브랜치를 통째로 넣을 수 있다는 것만 알면 된다. `master` 브랜치로 되돌아가서 `rack_branch`를 `rack` 디렉토리에 넣는다:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

그리고 나서 커밋을 하면 rack 디렉토리는 rack 프로젝트의 파일들을 직접 복사해 넣은 것과 똑같다. 복사한 것과 다른 점은 브랜치를 자유롭게 바꿀 수 있고 최신 버전의 Rack 프로젝트의 코드를 쉽게 끌어 올 수 있다는 점이다:

```
$ git checkout rack_branch
$ git pull
```

그리고 `git merge -s subtree`라는 명령어를 사용하여 `master` 브랜치와 Merge할 수 있고 원하든 원하지 않든 간에 히스토리도 함께 Merge된다. 수정 내용만 Merge하거나 커밋 메시지를 다시 작성하려면 `-s subtree` 옵션에다가 `--squash`, `--no-commit`를 함께 사용해야 한다:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Rack 프로젝트의 최신 코드를 가져다가 Merge했고 이제 커밋하면 된다. 물론 반대로 하는 것도 가능하다. `rack` 디렉토리로 이동해서 코드를 수정하고 `rack_branch` 브랜치로 Merge한다. 그리고 Rack 프로젝트 저장소에 Push할 수 있다.
`rack` 디렉토리와 `rack_branch` 브랜치와의 차이점도 비교할 수 있다. 일반적인 `diff` 명령은 사용할 수 없고 `git diff-tree` 명령을 사용해야 한다:

```
$ git diff-tree -p rack_branch
```

또 `rack` 디렉토리와 저장소의 `master` 브랜치와 비교할 수 있다:

```
$ git diff-tree -p rack_remote/master
```

6.8 요약

커밋과 저장소를 꼼꼼하게 관리하는 도구를 살펴보았다. 문제가 생기면 바로 누가, 언제, 무엇을 했는지 찾아내야 한다. 그리고 프로젝트를 쪼개고 싶을 때 사용하는 방법들도 배웠다. 이제 Git 명령어는 거의 모두 배운 것이다. 독자들이 하루빨리 익숙해져서 자유롭게 사용했으면 좋겠다.

7장

Git맞춤

지금까지 Git이 어떻게 동작하고 Git을 어떻게 사용하는지 설명했다. 이제 Git을 좀 더 쉽고 편하게 사용할 수 있도록 도와주는 도구를 살펴본다. 이 장에서는 먼저 많이 쓰이는 설정 그리고 훅 시스템을 먼저 설명한다. 그 후에 Git을 내게 맞추어(Customize) 본다. Git을 자신의 프로젝트에 맞추고 편하게 사용하자.

7.1 Git 설정하기

1장에서 설명했지만, 제일 먼저 해야 하는 것은 `git config` 명령으로 이름과 e-mail 주소를 설정하는 것이다:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

이렇게 설정하는 것들 중에서 중요한 것을 몇 가지 설명한다.

아주 기초적인 설정은1장에서도 설명했지만, 이번 장에서 다시 한 번 복습한다. Git은 내장된 기본 규칙 따르지만, 설정된 것이 있으면 그에 따른다. Git은 먼저 `/etc/gitconfig` 파일을 찾는다. 이 파일은 해당 시스템에 있는 모든 사용자와 모든 저장소에 적용되는 설정 파일이다. `git config` 명령에 `--system` 옵션을 주면 이 파일을 사용한다.

다음으로 `~/.gitconfig` 파일을 찾는다. 이 파일은 해당 사용자에게만 적용되는 설정 파일이다. `--global` 옵션을 주면 Git은 이 파일을 사용한다.

마지막으로 현재 작업 중인 저장소의 Git 디렉토리에 있는 `.git/config` 파일을 찾는다. 이 파일은 해당 저장소에만 적용된다. 각 설정 파일에 중복된 설정이 있으면 설명한 순서대로 덮어쓴다. 예를 들어 `.git/config`와 `/etc/gitconfig`에 같은 설정이 들어 있다면 `.git/config`에 있는 설정을 사용한다. 설정 파일은 손으로 직접 편집해도 되지만 보통 `git config` 명령을 사용하는 것이 더 편하다.

7.1.1 클라이언트 설정

설정은 클라이언트와 서버로 나뉜다. 대부분은 개인작업 환경과 관련된 클라이언트 설정이다. Git에는 설정거리가 매우 많은데, 여기서는 Workflow를 관리하는 데 필요한 것과 잘 사용하는 것만 설명한다. 한 번도 겪지 못할 상황에서나 유용한 옵션까지 다 포함하면 설정할 게 너무 많다. Git 버전마다 옵션이 조금씩 다른데, 아래와 같이 실행하면 설치한 버전에서 사용할 수 있는 옵션을 모두 보여준다:

```
$ git config --help
```

어떤 옵션을 사용할 수 있는지 `git config`의 에 자세히 설명돼 있다.

core.editor

Git은 편집기를 설정하지 않았거나 설정한 편집기를 찾을 수 없으면 Vi를 실행한다. 커밋할 때나 tag 메시지를 편집할 때 설정한 편집기를 실행한다. `code.editor` 설정으로 편집기를 설정한다:

```
$ git config --global core.editor emacs
```

이렇게 설정하면 메시지를 편집할 때 환경변수에 설정한 편집기가 아니라 Emacs를 실행한다.

commit.template

커밋할 때 Git이 보여주는 커밋 메시지는 이 옵션에 설정한 템플릿 파일이다. 예를 들어 `$HOME/.gitmessage.txt` 파일을 아래와 같이 만든다:

```
subject line

what happened

[ticket: X]
```

이 파일을 `commit.template`에 설정하면 Git은 `git commit` 명령이 실행하는 편집기에 이 메시지를 기본으로 넣어준다:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

그러면 commit할 때 아래와 같은 메시지를 편집기에 자동으로 채워준다:

```
subject line

what happened

[ticket: X]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
```

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~
~
".git/COMMIT_EDITMSG" 14L, 297C
```

소속 팀에 커밋 메시지 규칙이 있으면 그 규칙에 맞는 템플릿 파일을 만든다. Git이 그 파일을 사용하도록 설정하면 규칙을 따르기가 쉬워진다.

core.pager

Git은 `log`나 `diff`같은 명령의 메시지를 출력할 때 페이지로 나누어 보여준다. 기본으로 사용하는 명령은 `less`다. `more`를 더 좋아하면 `more`라고 설정한다. 페이지를 나누고 싶지 않으면 빈 문자열로 설정한다:

```
$ git config --global core.pager ''
```

이 명령을 실행하면 Git은 길든지 짧든지 결과를 한 번에 다 보여 준다.

user.signingkey

2장에서 설명했던 Annotated Tag를 만들 때 유용하다. 사용할 GPG 키를 설정해 둘 수 있다. 아래 처럼 GPG 키를 설정하면 서명할 때 편리하다:

```
$ git config --global user.signingkey <gpg-key-id>
```

`git tag` 명령을 실행할 때 키를 생략하고 서명할 수 있다:

```
$ git tag -s <tag-name>
```

core.excludesfile

Git이 무시하는 untracked 파일은 `.gitignore`에 해당 패턴을 적으면 된다고 2장에서 설명했다. 해당 패턴의 파일은 `git add` 명령으로 추가해도 Stage되지 않는다. `.gitignore` 파일을 저장소 밖에 두고 관리하고 싶으면 `core.excludesfile`에 해당 파일의 경로를 설정한다. 이 파일을 작성하는 방법은 `.gitignore` 파일을 작성하는 방법과 같다. 그리고 `core.excludesfile`에 설정한 파일과 저장소 안에 있는 `.gitignore` 파일은 둘 다 사용된다.

help.autocorrect

이 옵션은 Git 1.6.1 버전부터 사용할 수 있다. 명령어를 잘못 입력하면 Git은 메시지를 아래와 같이 보여 준다:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.

Did you mean this?
     commit
```

그러나 `help.autocorrect`를 1로 설정하면 명령어를 잘못 입력해도 Git이 자동으로 해당 명령어를 찾아서 실행해준다. 단, 해당 명령어가 딱 하나 찾았을 때에만 실행한다.

7.1.2 컬러 터미널

사람이 쉽게 인식할 수 있도록 터미널에 결과를 컬러로 출력할 수 있다. 터미널 컬러와 관련된 옵션은 매우 다양하기 때문에 꼼꼼하게 설정할 수 있다.

color.ui

`color.ui`를 true로 설정하면 Git이 알아서 결과에 색칠한다. 물론 무엇을 어떤 색으로 칠할지 꼼꼼하게 설정할 수 있지만, 이 옵션을 켜면 터미널을 그냥 기본 컬러로 칠한다.

```
$ git config --global color.ui true
```

이 옵션을 켜면 Git은 터미널에 컬러로 결과를 출력한다. 이 값을 false로 설정하면 절대 컬러로 출력하지 않는다. 결과를 파일로 리다이렉트하거나 다른 프로그램으로 보낼(Piping. 파이프라인)때도 그렇다.
`color.ui = always`라고 설정하면 결과를 리다이렉트할 때에도 컬러 코드가 출력된다. 이렇게까지 설정해야 하는 경우는 매우 드물다. 대신 Git 명령에는 `--color` 옵션이 있어서 어떻게 출력할지 그때그때 정해줄 수 있다. 보통은 `color.ui = true` 만으로도 충분하다.

`color.*`

Git은 좀 더 꼼꼼하게 컬러를 설정하는 방법을 제공한다. 아래와 같은 설정들이 있다. 모두 `true`, `false`, `always` 중 하나를 고를 수 있다:

```
color.branch
color.diff
color.interactive
color.status
```

또한, 각 옵션의 컬러를 직접 지정할 수도 있다. 아래처럼 설정하면 diff 명령에서 meta 정보의 포그라운드는 blue, 백그라운드는 black, 테스트는 bold로 바뀐다:

```
$ git config --global color.diff.meta "blue black bold"
```

컬러는 normal, black, red, green, yellow, blue, magenta, cyan, white 중에서 고를 수 있고 텍스트 속성은 bold, dim, ul, blink, reverse 중에서 고를 수 있다.
`git config` 맨페이지를 보면 어떤 설정거리가 있는지 자세히 나온다.

7.1.3 다른 Merge, Diff 도구 사용하기

Git에 들어 있는 diff 말고 다른 도구로 바꿀 수 있다. 화려한 GUI 도구로 바꿔서 좀 더 편리하게 충돌을 해결할 수 있다. 여기서는 Perforce의 Merge 도구인 P4Merge로 설정하는 것을 보여준다. P4Merge는 무료인데다 꽤 괜찮다.
P4Merge는 중요 플랫폼을 모두 지원하기 때문에 웬만한 환경이면 사용할 수 있다. 여기서는 Mac과 Linux 시스템에 설치하는 것을 보여준다. 윈도에서 사용하려면 `/usr/local/bin` 경로만 윈도 경로로 바꿔준다.
다음 페이지에서 P4Merge를 내려받는다:

```
http://www.perforce.com/product/components/perforce-visual-merge-and-diff-tools
```

먼저 P4Merge에 쓸 Wrapper 스크립트를 만든다. 필자는 Mac 사용자라서 Mac 경로를 사용한다. 어떤 시스템이든 `p4merge`가 설치된 경로를 사용하면 된다. extMerge라는 Merge용 Wrapper 스크립트를 만들고 이 스크립트로 넘어오는 모든 아규먼트를 p4merge 프로그램으로 넘긴다:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

그리고 diff용 Wrapper도 만든다. 이 스크립트로 넘어오는 아규먼트는 총 7개지만 그 중 2개만 Merge Wrapper로 넘긴다. Git이 diff 프로그램에 넘겨주는 아규먼트는 아래와 같다:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

이 중에서 `old-file`과 `new-file` 만 사용하는 wrapper script를 만든다:

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

이 두 스크립트에 실행 권한을 부여한다:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Git config 파일에 이 스크립트를 모두 추가한다. 설정해야 하는 옵션이 좀 많다. `merge.tool`로 무슨 Merge 도구를 사용할지, `mergetool.*.cmd`로 실제로 어떻게 명령어를 실행할지, `mergetool.trustExitCode`로 Merge 도구가 반환하는 exit 코드가 merge의 성공여부를 나타내는지, `diff.external`은 diff할 때 실행할 명령어가 무엇인지를 설정할 때 사용한다. 모두 `git config` 명령으로 설정한다:

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

`~/.gitconfig/` 파일을 직접 편집해도 된다:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff
```

설정을 완료하고 나서 아래와 같이 diff 명령어를 실행한다:

```
$ git diff 32d1776b1^ 32d1776b1
```

diff 결과가 터미널에 출력되는 대신 P4Merge가 실행된다. 그리고 그림 7-1처럼 그 프로그램 안에서 보여준다:

브랜치를 Merge할 때 충돌이 나면 `git mergetool` 명령을 실행한다. 이 명령을 실행하면 GUI 도구로 충돌을 해결할 수 있도록 P4Merge를 실행해준다.

Wrapper를 만들어 설정해두면 다른 diff, Merge 도구로 바꾸기도 쉽다. 예를 들어, KDiff3를 사용하도록 extDiff와 extMerge 스크립트를 수정한다:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

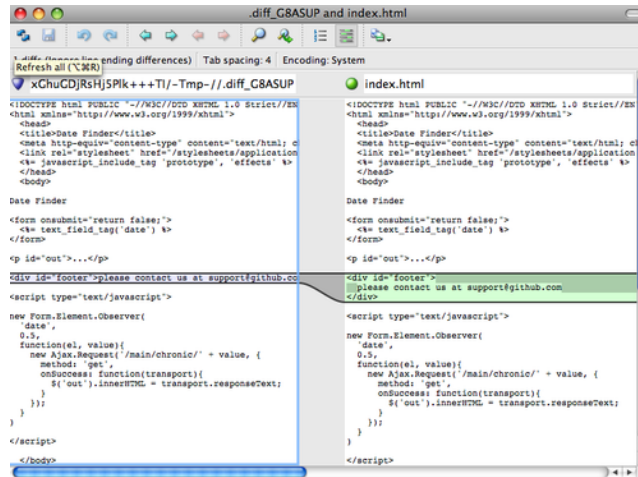



그림 7.1: P4Merge

이제부터 Git은 diff 결과를 보여주거나 충돌을 해결할 때 KDiff3 도구를 사용한다.
어떤 Merge 도구는 Git에 미리 cmd 설정이 들어 있다. 그래서 cmd 설정 없이 사용할 수 있는 것도 있다. kdiff3, opendiff, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff는 cmd 설정 없이 Merge 도구로 사용할 수 있다. diff 도구로는 다른 것을 사용하지만, Merge 도구로는 KDiff3를 사용하고 싶은 경우에는 kdiff3 명령을 실행경로로 넣고 아래와 같이 설정하기만 하면 된다:

```
$ git config --global merge.tool kdiff3
```

extMerge와 extDiff 파일을 사용하지 않고 이렇게 Merge 도구만 kdiff3로 설정하고 diff 도구는 Git에 원래 들어 있는 것을 사용할 수 있다.

7.1.4 소스 포맷과 공백

협업할 때 겪는 소스 포맷(Formatting)과 공백 문제는 미묘하고 난해하다. 동료 사이에 사용하는 플랫폼이 다를 때는 특히 더 심하다. 다른 사람이 보내온 Patch는 공백 문자 패턴이 미묘하게 다를 확률이 높다. 편집기가 몰래 공백문자를 추가해 버릴 수도 있고 크로스-플랫폼 프로젝트에서 윈도 개발자가 줄 끝에 CR(Carriage-Return) 문자를 추가해 버렸을 수도 있다. Git에는 이 이슈를 돕는 몇 가지 설정이 있다.

core.autocrlf

윈도에서 개발하는 동료와 함께 일하면 줄 바꿈(New Line) 문자에 문제가 생긴다. 윈도는 줄 바꿈 문자로 CR(Carriage-Return)과 LF(Line Feed) 문자를 둘 다 사용하지만, Mac과 Linux는 LF 문자만 사용한다. 아무것도 아닌 것 같지만, 크로스 플랫폼 프로젝트에서는 꽤 성가신 문제다.
Git은 커밋할 때 자동으로 CRLF를 LF로 변환해주고 반대로 Checkout할 때 LF를 CRLF로 변환해주는 기능이 있다. `core.autocrlf` 설정으로 이 기능을 켤 수 있다. 윈도에서 이 값을 true로 설정하면 Checkout할 때 LF 문자가 CRLR 문자로 변환된다:

```
$ git config --global core.autocrlf true
```

줄 바꿈 문자로 LF를 사용하는 Linux와 Mac에서는 Checkout할 때 Git이 LF를 CRLF로 변환할 필요가 없다. 게다가 우연히 CRLF가 들어간 파일이 저장소에 들어 있어도 Git이 알아서 고쳐주면 좋을 것이다. `core.autocrlf` 값을 input으로 설정하면 커밋할 때만 CRLF를 LF로 변환한다:

```
$ git config --global core.autocrlf input
```

이 설정을 이용하면 윈도에서는 CRLF를 사용하고 Mac, Linux, 저장소에서는 LF를 사용할 수 있다.
윈도 플랫폼에서만 개발하면 이 기능이 필요 없다. 이 옵션을 `false`라고 설정하면 이 기능이 꺼지고 CR 문자도 저장소에도 저장된다:

```
$ git config --global core.autocrlf false
```

core.whitespace

Git에는 공백 문자를 다루는 방법으로 네 가지가 미리 정의돼 있다. 두 가지는 기본적으로 켜져 있지만 끌 수 있고 나머지 두 가지는 꺼져 있지만 켤 수 있다.
먼저 기본적으로 켜져 있는 것을 살펴보자. `trailing-space`는 각 줄 끝에 공백이 있는지 찾고 `space-before-tab`은 모든 줄 처음에 tab보다 공백이 먼저 나오는지 찾는다.
기본적으로 꺼져 있는 나머지 두 개는 `indent-with-non-tab`과 `cr-at-eol`이다. `intent-with-non-tab`은 tab이 아니라 공백 8자 이상으로 시작하는 줄이 있는지 찾고 `cr-at-eol`은 줄 끝에 CR 문자가 있어도 괜찮다고 Git에 알리는 것이다.
`core.whitespace` 옵션으로 이 네 가지 방법을 켜고 끌 수 있다. 설정에서 해당 옵션을 빼버리거나 이름이 -로 시작하면 기능이 꺼진다. 예를 들어, 다른 건 다 켜고 `cr-at-eol` 옵션만 끄려면 아래와 같이 설정한다:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

`git diff` 명령을 실행하면 Git은 이 설정에 따라 검사해서 컬러로 표시해준다. 그래서 좀 더 쉽게 검토해서 커밋할 수 있다. `git apply` 명령으로 Patch를 적용할 때도 이 설정을 이용할 수 있다. 아래처럼 명령어를 실행하면 해당 Patch가 공백문자 정책에 들어맞는지 확인할 수 있다:

```
$ git apply --whitespace=warn <patch>
```

아니면 Git이 자동으로 고치도록 할 수 있다:

```
$ git apply --whitespace=fix <patch>
```

이 옵션은 `git rebase` 명령에서도 사용할 수 있다. 공백 문제가 있는 커밋을 서버로 Push하기 전에 `--whitespace=fix` 옵션을 주고 Rebase하면 Git은 다시 Patch를 적용하면서 공백을 설정한 대로 고친다.

7.1.5 서버 설정

서버 설정은 많지 않지만, 꼭 짚고 넘어가야 하는 것이 몇 개 있다.

receive.fsckObjects

Git은 Push할 때 기본적으로 개체를 검증하지(check for consistency) 않는다. 하지만, Push할 때마다 각 개체가 SHA-1 체크섬에 맞는지, 잘못된 개체가 가리키고 있는지 검사하게 할 수 있다. 개체를 점검하는 것은 상대적으로 느려서 Push하는 시간이 늘어난다. 얼마나 늘어나는지는 저장소 크기와 Push하는 양에 달렸다. `receive.fsckOBjects` 값을 true로 설정하면 Git이 Push할 때마다 검증한다.

```
$ git config --system receive.fsckObjects true
```

이렇게 설정하면 Push할 때마다 검증하기 때문에 클라이언트는 잘못된 데이터를 Push하지 못한다.

receive.denyNonFastForwards

이미 Push한 커밋을 Rebase해서 다시 Push하지 못하게 할 수 있다. 브랜치를 Push할 때 해당 리모트 브랜치가 가리키는 커밋이 Push하려는 브랜치에 없을 때 Push하지 못하게 할 수 있다. 보통은 이런 정책이 좋고 `git push` 명령에 `-f` 옵션을 주면 강제로 Push할 수 있다.
하지만, 강제로 Push하지 못하게 할 수도 있다. `receive.denyNonFastForwards` 옵션을 켜면 Fast-forward로 Push할 수 없는 브랜치는 아예 Push하지 못한다:

```
$ git config --system receive.denyNonFastForwards true
```

사용자마다 다른 정책을 적용하고 싶으면 서버 훅을 사용해야 한다. 서버의 receive 훅으로 할 수 있고 이 훅도 이 장에서 설명한다.

receive.denyDeletes

`receive.denyNonFastForwards`와 비슷한 정책으로 `receive.denyDeletes`라는 것이 있다. 이 설정을 켜면 브랜치를 삭제하는 Push가 거절된다. Git 1.6.1부터 receive.denyDeletes를 사용할 수 있다:

```
$ git config --system receive.denyDeletes true
```

이제 브랜치나 Tag를 삭제하는 Push는 거절된다. 아무도 삭제할 수 없다. 리모트 브랜치를 삭제하려면 직접 손으로 server의 ref 파일을 삭제해야 한다. 그리고 사용자마다 다른 정책을 적용시키는 ACL을 만드는 방법도 있다. 이 방법은 이 장 끝 부분에서 다룬다.

7.2 Git Attribute

디렉토리와 파일 단위로 다른 설정을 적용할 수도 있다. 이렇게 경로별로 설정하는 것을 'Git Attribute'라고 부른다. 이 설정은 `.gitattributes`라는 파일에 저장하고 아무 디렉토리에나 둘 수 있지만,

보통은 프로젝트 최상위 디렉토리에 둔다. 그리고 이 파일을 커밋하고 싶지 않으면 `.gitattributes`가 아니라 `.git/info/attributes`로 파일을 만든다.
이 Attribute로 Merge는 어떻게 할지, 텍스트가 아닌 파일은 어떻게 Diff할지, checkin/checkout 할 때 어떻게 필터링할지 정해줄 수 있다. 이 절에서는 설정할 수 있는 Attribute가 어떤 것이 있는지, 그리고 어떻게 설정하는지 배우고 예제를 살펴본다.

7.2.1 바이너리 파일

이 Attribute로 어떤 파일이 바이너리 파일인지 Git에게 알려줄 수 있다. 기본적으로 Git은 어떤 파일이 바이너리 파일인지 알지 못한다. 하지만, Git에는 파일을 어떻게 다뤄야 하는지 알려주는 방법이 있다. 텍스트 파일 중에서 프로그램이 생성하는 파일에는 바이너리 파일과 진배없는 파일이 있다. 이런 파일은 diff할 수 없으니 바이너리 파일이라고 알려줘야 한다. 반대로 바이너리 파일 중에서 취급 방법을 Git에 알려주면 diff할 수 있는 파일도 있다.

바이너리 파일이라고 알려주기

사실 텍스트 파일이지만 만든 목적과 의도를 보면 바이너리 파일인 것이 있다. 예를 들어 Mac의 Xcode는 `.pbxproj` 파일을 만든다. 이 파일은 IDE 설정 등을 디스크에 저장하는 파일로 JSON 포맷이다. 모든 것이 ASCII인 텍스트 파일이지만 실제로는 간단한 데이터베이스이기 때문에 텍스트 파일처럼 취급할 수 없다. 그래서 여러 명이 이 파일을 동시에 수정하고 Merge할 때 diff가 도움이 안 된다. 이 파일은 프로그램이 읽고 쓰는 파일이기 때문에 바이너리 파일처럼 취급하는 것이 옳다.
모든 `pbxproj` 파일을 바이너리로 파일로 취급하는 설정은 아래와 같다. `.gitattributes` 파일에 넣으면 된다:

```
*.pbxproj -crlf -diff
```

이제 `pbxproj` 파일은 CRLF 변환이 적용되지 않는다. `git show`나 `git diff` 같은 명령을 실행할 때에도 통계를 계산하거나 diff를 출력하지 않는다. Git 1.6부터는 `-crlf -diff`를 한 마디로 줄여서 표현할 수 있다:

```
*.pbxproj binary
```

바이너리 파일 Diff하기

Git은 바이너리 파일도 diff할 수 있다. Git Attribute를 통해 Git이 바이너리 파일을 텍스트 포맷으로 변환하고 그 결과를 diff로 비교하도록 하는 것이다. 그래서 문제는 어떻게 바이너리를 텍스트로 변환해야 할까에 있다. 바이너리를 텍스트로 변환해 주는 도구 중에서 내가 필요한 바이너리 파일에 꼭 맞는 도구를 찾는게 가장 좋다. 사람이 읽을 수 있는 텍스트로 표현된 바이너리 포맷은 극히 드물다(오디오 데이터를 텍스트로 변환한다고 생각해보라). 파일 내용을 텍스트로 변환할 방법을 찾지 못했을 때는 파일의 설명이나 메타데이터를 텍스트로 변환하는 방법을 찾아보자. 이런 방법이 가능한 경우가 많다. 메타데이터는 파일 내용을 완벽하게 알려주지 않지만 전혀 비교하지 못하는 것보다 이렇게라도 하는 게 훨씬 낫다.
여기서 설명한 두 가지 방법을 많이 사용하는 바이너리 파일에 적용해 볼 거다.

댓글: 전용 변환기는 없지만 텍스트가 들어 있는 바이너리 포맷들이 있다. 이런 포맷은 `strings` 프로그램으로 바이너리 파일에서 텍스트를 추출한다. 이런 종류의 바이너리 파일 중에서 UTF-16 인코딩이나 다른 "codepages"로 된 파일들도 있다. 그런 인코딩으로 된 파일에서 `strings`으로 추출할 수 있는 텍스트는 제한적이다. 상황에 따라 다르게 추출된다. 그래도 `strings`는 Mac과 Linux 시스템에서 쉽게 사용할 수 있기 때문에 다양한 바이너리 파일에 쉽게 적용할 수 있다.

MS Word 파일 먼저 이 기술을 인류에게 알려진 가장 귀찮은 문제 중 하나인 Word 문서를 버전 관리하는 상황을 살펴보자. 모든 사람이 Word가 가장 끔찍한 편집기라고 말하지만 애석하게도 모두 Word를 사용한다. Git 저장소에 넣고 이따금 커밋하는 것만으로도 Word 문서의 버전을 관리할 수 있다. 그렇지만 `git diff`를 실행하면 다음과 같은 메시지를 볼 수 있을 뿐이다:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

직접 파일을 하나하나 까보지 않으면 두 버전이 뭐가 다른지 알 수 없다. Git Attribute를 사용하면 이를 더 좋게 개선할 수 있다. `.gitattributes` 파일에 아래와 같은 내용을 추가한다:

```
*.doc diff=word
```

이것은 `*.doc` 파일의 두 버전이 무엇이 다른지 diff할 때 "word" 필터를 사용하라고 설정하는 것이다. 그럼 "word" 필터는 뭘까? 이 "word" 필터도 정의해야 한다. Word 문서에서 사람이 읽을 수 있는 텍스트를 추출해주는 `catdoc` 프로그램을 "word" 필터로 사용한다. 그러면 Word 문서를 diff할 수 있다. (catdoc 프로그램은 MS Word 문서에 특화된 텍스트 추출기다. `http://www.wagner.pp.ru/~vitus/software/catdoc/`에서 구할 수 있다):

```
$ git config diff.word.textconv catdoc
```

위의 명령은 아래와 같은 내용을 `.git/config` 파일에 추가한다:

```
[diff "word"]
    textconv = catdoc
```

이제 Git은 확장자가 `.doc`인 파일의 스냅샷을 diff할 때 "word" 필터로 정의한 `catdoc` 프로그램을 사용한다. 이 프로그램은 Word 파일을 텍스트 파일로 변환해 주기 때문에 diff할 수 있다.

이 책의 1장을 Word 파일로 만들어서 Git에 넣고 나서 단락 하나를 수정하고 저장하는 예를 살펴보자. `git diff`를 실행하면 어디가 달려졌는지 확인할 수 있다:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -128,7 +128,7 @@ and data size)
 Since its birth in 2005, Git has evolved and matured to be easy to use
 and yet retain these initial qualities. It’s incredibly fast, it’s
 very efficient with large projects, and it has an incredible branching
-system for non-linear development.
+system for non-linear development (See Chapter 3).
```

Git은 “(See Chapter 3)”가 추가됐다는 것을 정확하게 찾아 준다.

OpenDocument 파일 MS Word(`*.doc`) 파일에 사용한 방법은 OpenOffice.org(혹은 LibreOffice.org) 파일 형식인 OpenDocument(`*.odt`) 파일에도 적용할 수 있다.
아래의 내용을 `.gitattributes` 파일에 추가한다:

```
*.odt diff=odt
```

`.git/config` 파일에 `odt` diff 필터를 설정한다:

```
[diff "odt"]
    binary = true
    textconv = /usr/local/bin/odt-to-txt
```

OpenDocument 파일은 사실 여러 파일(XML, 스타일, 이미지 등등)을 Zip으로 압축한 형식이다. OpenDocument 파일에서 텍스트만 추출하는 스크립트를 하나 작성한다. 아래와 같은 내용을 `/usr/local/bin/odt-to-txt` 파일로(다른 위치에 저장해도 상관없다) 저장한다:

```
#! /usr/bin/env perl
# Simplistic OpenDocument Text (.odt) to plain text converter.
# Author: Philipp Kempgen

if (! defined($ARGV[0])) {
    print STDERR "No filename given!\n";
    print STDERR "Usage: $0 filename\n";
    exit 1;
}
```

```
my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
{
    local $/ = undef;  # slurp mode
    $content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^>]*>//g;           # remove spans
s/<text:h\b[^>]*>/\n\n*****  /g;   # headers
s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>/\n    --  /g;  # list items
s/<text:list\b[^>]*>/\n\n/g;       # lists
s/<text:p\b[^>]*>/\n  /g;          # paragraphs
s/<[^>]+>//g;                      # remove all XML tags
s/\n{2,}/\n\n/g;                   # remove multiple blank lines
s/\A\n+//;                         # remove leading blank lines
print "\n", $_, "\n\n";
```

그리고 실행 가능하도록 만든다:

```
chmod +x /usr/local/bin/odt-to-txt
```

이제 `git diff` 명령으로 `.odt` 파일에 대한 변화를 살펴볼 수 있다.

이미지 파일 이 방법으로 이미지 파일도 diff할 수 있다. 필터로 EXIF 정보를 추출해서 PNG 파일을 비교한다. EXIF 정보는 대부분의 이미지 파일에 들어 있는 메타데이터다. `exiftool`이라는 프로그램을 설치하고 이미지 파일에서 메타데이터 텍스트를 추출한다. 그리고 그 결과를 diff해서 무엇이 달라졌는지 본다:

```
$ echo '*.pngdiff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

프로젝트에 들어 있는 이미지 파일을 변경하고 `git diff`를 실행하면 아래와 같이 보여준다:

```
diff --git a/image.pngb/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
```

```
 ExifTool Version Number         : 7.74
-File Size                       : 70 kB
-File Modification Date/Time     : 2009:04:17 10:12:35-07:00
+File Size                       : 94 kB
+File Modification Date/Time     : 2009:04:21 07:02:43-07:00
 File Type                       : PNG
 MIME Type                       : image/png
-Image Width                     : 1058
-Image Height                    : 889
+Image Width                     : 1056
+Image Height                    : 827
 Bit Depth                       : 8
 Color Type                      : RGB with Alpha
```

이미지 파일의 크기와 해상도가 달라진 것을 쉽게 알 수 있다:

7.2.2 키워드 치환

SVN이나 CVS에 익숙한 사람들은 해당 시스템에서 사용하던 키워드 치환(Keyword Expansion) 기능을 찾는다. Git에서는 이것이 쉽지 않다. Git은 먼저 체크섬을 계산하고 커밋하기 때문에 그 커밋에 대한 정보를 가지고 파일을 수정할 수 없다. 하지만, Checkout할 때 그 정보가 자동으로 파일에 삽입되도록 했다가 다시 커밋할 때 삭제되도록 할 수 있다.

파일 안에 `$Id$` 필드를 넣으면 Blob의 SHA-1 체크섬을 자동으로 삽입한다. 이 필드를 파일에 넣으면 Git은 앞으로 Checkout할 때 해당 Blob의 SHA-1 값으로 교체한다. 여기서 꼭 기억해야 할 것이 있다. 교체되는 체크섬은 커밋의 것이 아니라 Blob 그 자체의 SHA-1 체크섬이다:

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

Git은 이 파일을 Checkout할 때마다 SHA 값을 삽입해준다:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

하지만 이것은 별로 유용하지 않다. CVS나 SVN의 키워드 치환(Keyword Substitution)을 써봤으면 날짜(Datestamp)도 가능했다는 것을 알고 있을 것이다. SHA는 그냥 해시이고 식별할 수 있을 뿐이지 다른 것을 알려주진 않는다. SHA만으로는 예전 것보다 새 것인지 오래된 것인지는 알 수 없다.

Commit/Checkout할 때 사용하는 필터를 직접 만들어 쓸 수 있다. 방향에 따라 “clean” 필터와 “smudge” 필터라고 부른다. “.gitattributes” 파일에 설정하고 파일 경로마다 다른 필터를 설정할 수

있다. Checkout할 때 파일을 처리하는 것이 "smudge" 필터이고(그림 7-2) 커밋할 때 처리하는 필터가 "clean" 필터이다. 이 필터로 할 수 있는 일은 무궁무진하다.

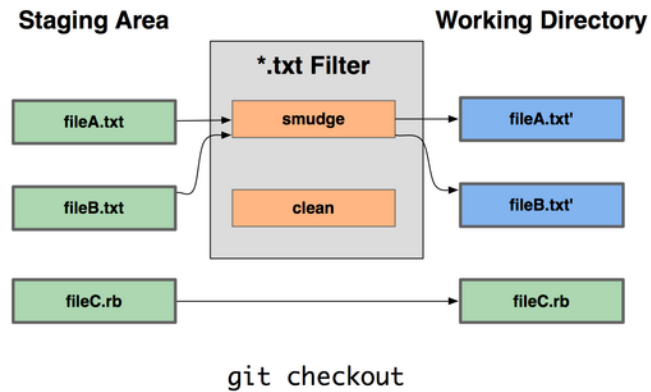



그림 7.2: "smudge" 필터는 Checkout할 때 실행된다

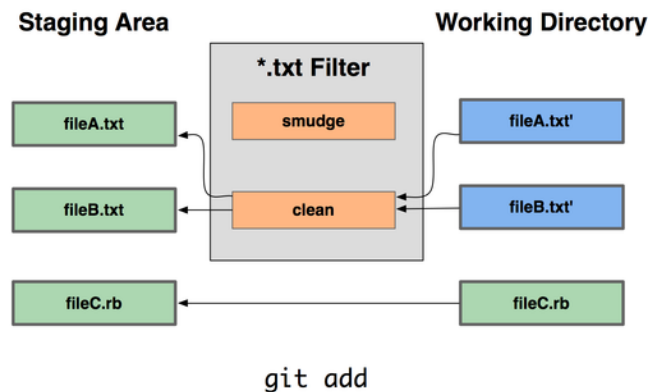



그림 7.3: "clean" 필터는 파일을 Stage할 때 실행된다

커밋하기 전에 `indent` 프로그램으로 C 코드 전부를 필터링하지만 커밋 메시지는 단순한 예제를 보자. `*.c` 파일은 indent 필터를 사용하도록 `.gitattributes` 파일에 설정한다:

```
*.c     filter=indent
```

아래처럼 "indent" 필터의 smudge와 clean이 무엇인지 설정한다:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

`*.c` 파일을 커밋하면 `indent` 프로그램을 통해서 커밋되고 Checkout하면 `cat` 프로그램을 통해 Checkout된다. `cat`은 입력된 데이터를 그대로 다시 내보내는, 사실 아무것도 안 하는 프로그램이다. 이렇게 설정하면 모든 C 소스 파일은 `indent` 프로그램을 통해 커밋된다.

이제 RCS 처럼 `$Date$`를 치환하는 예제을 살펴보자. 이것를 하려면 간단한 스크립트가 하나 필요하다. 이 스크립트는 `$Date$` 필드를 프로젝트의 마지막 커밋 일자로 치환한다. 표준 입력을 읽어서 `$Date$` 필드를 치환한다. 아래는 Ruby로 구현한 스크립트다:

```
#! /usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

`git log` 명령으로 마지막 커밋 정보를 얻고 표준 입력(STDIN)에서 `$Date$` 스트링을 찾아서 치환한다. 스크립트는 자신이 편한 언어로 만든다. 이 스크립트의 이름을 `expand_date`라고 짓고 실행 경로에 넣는다. 그리고 `dater`라는 Git 필터를 정의한다. Checkout시 실행하는 smudge 필터로 `expand_date`를 사용하고 커밋할 때 실행하는 clean 필터는 Perl을 사용한다:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\\\$]*\\\$/\\\$Date\\\$/"'
```

이 Perl 코드는 `$Date$` 스트링에 있는 문자를 제거해서 원래대로 복원한다. 이제 필터가 준비됐으니 `$Date$` 키워드가 들어 있는 파일을 만들고 Git Attribute를 설정한다. 새 필터를 시험해보자:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

커밋하고 파일을 다시 Checkout 하면 해당 키워드가 적절히 치환된 것을 볼 수 있다:

```
$ git add date_test.txt .gitattributes
$ git commit -m "Testing date expansion in Git"
$ rm date_test.txt
$ git checkout date_test.txt
$ cat date_test.txt
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

이것은 매우 강력해서 두루두루 사용할 수 있다. `.gitattributes` 파일은 커밋하는 파일이기 때문에 드라이버(여기서는 `dater`)가 없는 사람에게도 배포된다. 그리고 `dater`가 없으면 에러가 난다. 필터를 만들 때 이런 예외 상황도 고려해서 항상 잘 동작하게 해야 한다.

7.2.3 저장소 익스포트하기

프로젝트를 익스포트해서 아카이브를 만들 때에도 Git Attribute가 유용하다.

export-ignore

아카이브를 만들 때 제외할 파일이나 디렉토리가 무엇인지 설정할 수 있다. 특정 디렉토리나 파일을 프로젝트에는 포함하고 아카이브에는 포함하고 싶지 않을 때 `export-ignore` Attribute를 사용한다.

예를 들어 `test/` 디렉토리에 테스트 파일이 있다고 하자. 보통 tar 파일로 묶어서 익스포트할 때 테스트 파일은 포함하지 않는다. Git Attribute 파일에 다음 라인을 추가하면 테스트 파일은 무시된다:

```
test/ export-ignore
```

`git archive` 명령으로 tar 파일을 만들면 test 디렉토리는 아카이브에 포함되지 않는다.

export-subst

아카이브를 만들 때에도 키워드 치환을 할 수 있다. 파일을 하나 만들고 거기에 `$Format:$` 스트링을 넣으면 Git이 치환해준다. 이 스트링에 `--pretty=format` 옵션에 사용하는 것과 같은 포맷 코드를 넣을 수 있다. `--pretty=format`은 2장에서 배웠다. 예를 들어 `LAST_COMMIT`이라는 파일을 만들고 `git archive` 명령을 실행할 때 자동으로 이 파일에 마지막 커밋 날짜가 삽입되게 하려면 아래와 같이 해야 한다:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

`git archive` 명령으로 아카이브를 만들고 나서 이 파일을 열어보면 아래와 같이 보인다:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

7.2.4 Merge 전략

파일마다 다른 Merge 전략을 사용하도록 설정할 수 있다. Merge할 때 충돌이 날 것 같은 파일이 있다고 하자. Git Attrbute로 이 파일만 항상 타인의 코드 말고 내 코드를 사용하도록 설정할 수 있다.
이 설정은 다양한 환경에서 운영하려고 만든 환경 브랜치를 Merge할 때 좋다. 이때는 환경 설정과 관련된 파일은 Merge하지 않고 무시하는 게 편리하다. 브랜치에 `database.xml`이라는 데이터베이스 설정 파일이 있는데 이 파일은 브랜치마다 다르다. Database 설정 파일은 Merge하면 안된다. Attribute를 아래와 같이 설정하면 이 파일은 그냥 두고 Merge한다.

```
database.xml merge=ours
```

이제 Merge해도 `database.xml` 파일은 충돌하지 않는다:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

Merge했지만 `database.xml`은 원래 가지고 있던 파일 그대로다.

7.3 Git 훅

Git도 다른 버전 관리 시스템처럼 어떤 이벤트가 생겼을 때 자동으로 특정 스크립트를 실행하도록 할 수 있다. 이 훅은 클라이언트 훅과 서버 훅으로 나눌 수 있다. 클라이언트 훅은 커밋이나 Merge할 때 실행되고 서버 훅은 Push할 때 서버에서 실행된다. 이 절에서는 어떤 훅이 있고 어떻게 사용하는지 배운다.

7.3.1 훅 설치하기

훅은 Git 디렉토리 밑에 `hooks`라는 디렉토리에 저장한다. 기본 훅 디렉토리는 `.git/hooks`이다. 이 디렉토리에 가보면 Git이 자동으로 넣어준 매우 유용한 스크립트 예제가 몇 개 있다. 그리고 스크립트가 입력받는 값이 어떤 값인지 파일 안에 자세히 설명돼 있다. 모든 예제는 쉘과 Perl 스크립트로 작성돼 있지만 실행할 수만 있으면 되고 Ruby나 Python같은 다른 스크립트 언어로 만들어도 된다.예제 스크립트의 파일 이름에는 `.sample`이라는 확장자가 붙어 있다. 그래서 이름만 바꿔주면 그 훅을 사용할 수 있다.
실행할 수 있는 스크립트 파일을 저장소의 `hooks` 디렉토리에 넣으면 훅 스크립트가 켜진다. 이 스크립트는 앞으로 계속 호출된다. 중요한 훅은 여기서 모두 설명한다.

7.3.2 클라이언트 훅

클라이언트 훅은 매우 다양하다. 이 절에서는 클라이언트 훅을 커밋 Workflow 훅, E-mail Workflow 훅, 그리고 나머지로 분류해서 설명한다.

커밋 Workflow 훅

먼저 커밋과 관련된 훅을 살펴보자. 커밋과 관련된 훅은 모두 네 가지다. `pre-commit` 훅은 커밋할 때 가장 먼저 호출되는 훅으로 커밋 메시지를 작성하기 전에 호출된다. 이 훅에서 커밋하는 Snapshot을 점검한다. 빠트린 것은 없는지, 테스트는 확실히 했는지 등을 검사한다. 커밋할 때 꼭 확인해야 할 게 있으면 이 훅으로 확인한다. 그리고 이 훅의 Exit 코드가 0이 아니면 커밋은 취소된다. 물론 `git commit --no-verify`라고 실행하면 이 훅을 일시적으로 생략할 수 있다. lint 같은 프로그램으로 코드 스타일을 검사하거나, 줄 끝의 공백 문자를 검사하거나(예제로 들어 있는 `pre-commit` 훅이 하는 게 이 일이다), 코드에 주석을 달았는지 검사하는 일은 이 훅으로 하는 것이 좋다.
`prepare-commit-msg` 훅은 Git이 커밋 메시지를 생성하고 나서 편집기를 실행하기 전에 실행된다. 이 훅은 사람이 커밋 메시지를 수정하기 전에 먼저 프로그램으로 손보고 싶을 때 사용한다. 이 훅은 커밋 메시지가 들어 있는 파일의 경로, 커밋의 종류를 아규먼트로 받는다. 그리고 최근 커밋을 수정할 때에는(Amending 커밋) SHA-1 값을 추가 아규먼트로 더 받는다. 사실 이 훅은 일반 커밋에는 별로 필요 없고 커밋 메시지를 자동으로 생성하는 커밋에 좋다. 커밋 메시지에 템플릿을 적용하거나, Merge 커밋, Squash 커밋, Amend 커밋일 때 유용하다. 이 스크립트로 커밋 메시지 템플릿에 정보를 삽입할 수 있다.
`commit-msg` 훅은 커밋 메시지가 들어 있는 임시 파일의 경로를 아규먼트로 받는다. 그리고 이 스크립트가 0이 아닌 값을 반환하면 커밋되지 않는다. 이 훅에서 최종적으로 커밋이 완료되기 전에 프로젝트 상태나 커밋 메시지를 검증한다. 이 장의 마지막 절에서 이 훅을 사용하는 예제를 보여준다. 커밋 메시지가 정책에 맞는지 검사하는 스크립트를 만들어 보자.
커밋이 완료되면 `post-commit` 훅이 실행된다. 이 훅은 넘겨받는 아규먼트가 하나도 없지만 `git log -1 HEAD` 명령으로 정보를 쉽게 가져올 수 있다. 일반적으로 이 스크립트는 커밋된 것을 누군가에게 알릴 때 사용한다.

이 커밋 Workflow 스크립트는 어떤 Workflow에나 사용할 수 있다. 특히 정책을 강제할 때 유용하다. 클라이언트 훅은 개발자가 클라이언트에서 사용하는 훅이다. 모든 개발자에게 유용한 훅이지만 Clone할 때 복사되지 않는다. 그래서 직접 설치하고 관리해야 한다. 물론 정책을 서버 훅으로 만들고 정책을 잘 지키는지 Push할 때 검사해도 된다.

E-mail Workflow 훅

E-mail Workflow에 해당하는 클라이언트 훅은 세 가지이다. 이 훅은 모두 `git am` 명령으로 실행된다. 이 명령어를 사용할 일이 없으면 이 절은 읽지 않아도 된다. 하지만, 언젠가는 `git format-patch` 명령으로 만든 Patch를 E-mail로 받는 날이 올지도 모른다.

제일 먼저 실행하는 훅은 `applypatch-msg`이다. 이 훅의 아규먼트는 Author가 보내온 커밋 메시지 파일의 이름이다. 이 스크립트가 종료할 때 0이 아닌 값을 반환하면 Git은 Patch하지 않는다. 커밋 메시지가 규칙에 맞는지 확인하거나 자동으로 메시지를 수정할 때 이 훅을 사용한다.

`git am`으로 Patch할 때 두 번째로 실행되는 훅이 `pre-applypatch`이다. 이 훅은 아규먼트가 없고 단순히 Patch를 적용하고 나서 실행된다. 그래서 커밋할 스냅샷을 검사하는 데 사용한다. 이 스크립트로 테스트를 수행하고 파일을 검사할 수 있다. 테스트에 실패하거나 뭔가 부족하면 0이 아닌 값을 반환시켜서 `git am` 명령을 취소시킬 수 있다.

`git am` 명령에서 마지막으로 실행되는 훅은 `post-applypatch`다. 이 스크립트를 이용하면 자동으로 Patch를 보낸 사람이나 그룹에게 알림 메시지를 보낼 수 있다. 이 스크립트로는 Patch를 중단시킬 수 없다.

기타 훅

`pre-rebase` 훅은 Rebase하기 전에 실행된다. 이 훅이 0이 아닌 값을 반환하면 Rebase가 취소된다. 이 훅으로 이미 Push한 커밋을 Rebase하지 못하게 할 수 있다. Git이 자동으로 넣어주는 `pre-rebase` 예제가 바로 그 예제다. 이 예제에는 기준 브랜치가 `next`라고 돼 있다. 실제로 적용할 브랜치 이름으로 사용하면 된다.

그리고 `git checkout` 명령이 끝나면 `post-checkout` 훅이 실행된다. 이 훅은 Checkout할 때마다 작업하는 디렉토리에서 뭔가 할 일이 있을 때 사용한다. 그러니까 용량이 크거나 Git이 관리하지 않는 파일을 옮기거나, 문서를 자동으로 생성하는 데 쓴다.

마지막으로, `post-merge` 훅은 Merge가 끝나고 나서 실행된다. 이 훅은 파일 권한 같이 Git이 추적하지 않는 정보를 관리하는 데 사용한다. Merge로 Working Tree가 변경될 때 Git이 관리하지 않는 파일이 원하는 대로 잘 배치됐는지 검사할 때도 좋다.

7.3.3 서버 훅

클라이언트 훅으로도 어떤 정책을 강제할 수 있지만, 시스템 관리자에게는 서버 훅이 더 중요하다. 서버 훅은 모두 Push 전후에 실행된다. Push 전에 실행되는 훅이 0이 아닌 값을 반환하면 해당 Push는 거절되고 클라이언트는 에러 메시지를 출력한다. 이 훅으로 아주 복잡한 Push 정책도 가능하다.

pre-receive와 post-receive

Push하면 가장 처음 실행되는 훅은 `pre-receive` 훅이다. 이 스크립트는 표준 입력(STDIN)으로 Push하는 레퍼런스의 목록을 입력받는다. 0이 아닌 값을 반환하면 해당 레퍼런스가 전부 거절된다. Fast-forward Push가 아니면 거절하거나, 브랜치 Push 권한을 제어하려면 이 훅에서 하는 것이 좋다. 관리자만 브랜치를 새로 Push하고 삭제할 수 있고 일반 개발자는 수정사항만 Push할 수 있게 할 수 있다.

`post-receive` 훅은 Push한 후에 실행된다. 이 훅으로 사용자나 서비스에 알림 메시지를 보낼 수 있다. 그리고 `pre-receive` 훅처럼 표준 입력(STDIN)으로 레퍼런스 목록이 넘어간다. 이 훅으로 메일링리스트에 메일을 보내거나, CI(Continuous Integration) 서버나 Ticket-tracking 시스템의 정보를 수정할 수 있다. 심지어 커밋 메시지도 파싱할 수 있기 때문에 이 훅으로 Ticket을 만들고, 수정하고, 닫을 수 있다. 이 스크립트가 완전히 종료할 때까지 클라이언트와의 연결은 유지되고 Push를 중단시킬 수 없다. 그래서 이 스크립트로 시간이 오래 걸릴만한 일을 할 때는 조심해야 한다.

update

update 스크립트는 각 브랜치마다 한 번씩 실행된다는 것을 제외하면 `pre-receive` 스크립트와 거의 같다. 한 번에 브랜치를 여러 개 Push하면 `pre-receive`는 딱 한 번만 실행되지만, update는 브랜치마다 실행된다. 이 스크립트는 표준 입력으로 데이터를 입력받는 것이 아니라 아규먼트로 브랜치 이름, 원래 가리키던 SHA-1 값, 사용자가 Push하는 SHA-1 값을 입력받는다. update 스크립트가 0이 아닌 값을 반환하면 해당 레퍼런스만 거절되고 나머지 다른 레퍼런스는 상관없다.

7.4 정책 구현하기

지금까지 배운 것을 한 번 적용해보자. 커밋 메시지 규칙 검사하고, Fast-forward Push만 허용하고, 디렉토리마다 사용자의 수정 권한을 제어하는 Workflow를 만든다. 실질적으로 정책을 강제하려면 서버 훅으로 만들어야 한다. 하지만, 개발자들이 Push할 수 없는 커밋은 아예 만들지 않도록 클라이언트 훅도 만든다.

필자가 제일 좋아하는 Ruby로 만든다. 필자는 독자가 슈도코드를 읽듯이 Ruby 코드를 읽을 수 있다고 생각한다. Ruby를 모르더라도 충분히 개념을 이해할 수 있을 것이다. 하지만, Git은 언어를 가리지 않는다. Git이 자동으로 생성해주는 예제는 모두 Perl과 Bash로 작성돼 있다. 그래서 예제를 열어 보면 Perl과 Bash로 작성된 예제를 참고 할 수 있다.

7.4.1 서버 훅

서버 정책은 전부 update 훅으로 만든다. 이 스크립트는 브랜치가 Push될 때마다 한 번 실행되고 해당 브랜치의 이름, 원래 브랜치가 가리키던 레퍼런스, 새 레퍼런스를 아규먼트로 받는다. 그리고 SSH를 통해서 Push하는 것이라면 누가 Push하는 지도 알 수 있다. SSH로 접근하긴 하지만 개발자 모두 계정 하나로("git" 같은) Push하고 있다면 실제로 Push하는 사람이 누구인지 판별해주는 쉘 Wrapper가 필요하다. 이 스크립트에서는 `$USER` 환경 변수에 현재 접속한 사용자 정보가 있다고 가정한다. update 스크립트는 필요한 정보를 수집하는 것으로 시작한다:

```
#!/usr/bin/env ruby

refname = ARGV[0]
oldrev  = ARGV[1]
newrev  = ARGV[2]
user    = ENV['USER']

puts "Enforcing Policies... \n(#{refname}) (#{oldrev[0,6]}) (#{newrev[0,6]})"
```

커밋 메시지 규칙 만들기

커밋 메시지 규칙부터 해보자. 일단 목표가 있어야 하니까 커밋 메시지에 “ref: 1234” 같은 스트링이 포함돼 있어야 한다고 가정하자. 보통 커밋은 이슈 트래커에 있는 이슈와 관련돼 있으니 그 이슈가 뭔지 커밋 메시지에 적어 놓으면 좋다. Push할 때마다 커밋 메시지에 해당 스트링이 포함돼 있는지 확인한다. 만약 커밋 메시지에 해당 스트링이 없는 커밋이면 0이 아닌 값을 반환해서 Push를 거절한다.

`$newrev`, `$oldrev` 변수와 `git rev-list`라는 Plumbing 명령어를 이용해서 Push하는 커밋의 모든 SHA-1 값을 알 수 있다. 이것은 `git log`와 근본적으로 같은 명령이고 옵션을 하나도 주지 않으면 다른 정보 없이 SHA-1 값만 보여준다. 이 명령으로 Push하는 커밋을 모두 알 수 있다:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

이 SHA-1 값으로 각 커밋의 메시지도 가져온다. 커밋 메시지를 가져와서 정규표현식으로 해당 패턴이 있는지 검사한다.

커밋 메시지를 얻는 방법을 알아보자. 커밋의 raw 데이터는 `git cat-file`이라는 Plumbing 명령어로 얻을 수 있다. 9장에서 Plumbing 명령어에 대해 자세히 다루니까 지금은 커밋 메시지 얻는 것에 집중하자:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

이 명령이 출력하는 메시지에서 커밋 메시지만 잘라내야 한다. 첫 번째 빈 줄 다음부터가 커밋 메시지니까 유닉스 명령어 `sed`로 첫 빈 줄 이후를 잘라낸다.

```
$ git cat-file commit ca82a6 | sed '1,/^$/d'
changed the version number
```

이제 커밋 메시지에서 찾는 패턴과 일치하는 문자열이 있는지 검사해서 있으면 통과시키고 없으면 거절한다. 스크립트가 종료할 때 0이 아닌 값을 반환하면 Push가 거절된다. 이 일을 하는 코드는 아래와 같다:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

이 코드를 update 스크립트에 넣으면 규칙을 어긴 커밋은 Push할 수 없다.

ACL로 사용자마다 다른 규칙 적용하기

진행하는 프로젝트에 모듈이 여러 개 있는데, 모듈마다 속한 사용자들만 Push할 수 있게 설정해야 한다고 가정하자. 모든 권한을 다 가진 사람들도 있고 특정 디렉토리나 파일만 Push할 수 있는 사람도 있다. 이런 일을 강제하려면 먼저 서버의 Bare 저장소에 `acl`이라는 파일을 만들고 거기에 규칙을 기술한다. 그리고 update 훅에서 Push하는 파일이 무엇인지 확인하고 ACL과 비교해서 Push할 수 있는지 없는지 결정한다.

우선 ACL부터 작성한다. CVS에서 사용하는 것과 비슷한 ACL을 만든다. 규칙은 한 줄에 하나씩 기술한다. 각 줄의 첫 번째 필드는 `avail`이나 `unavail`이고 두 번째 필드는 규칙을 적용할 사용자들의 목록을 CSV(Comma-Separated Values) 형식으로 적는다. 마지막 필드엔 규칙을 적용할 경로를 적는다. 만약 마지막 필드가 비워져 있으면 모든 경로를 의미한다. 이 필드는 파이프(|) 문자로 구분한다.

관리자도 여러 명이고, doc 디렉토리에서 문서를 만드는 사람도 여러 명이다. 하지만 `lib`과 `tests` 디렉토리에 접근하는 사람은 한 명이다. 이런 상황을 ACL로 만들면 아래와 같다:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

이 ACL 정보는 스크립트에서 읽어 사용한다. 설명을 쉽게 하고자 여기서는 `avail`만 처리한다. 다음 메소드는 Associative Array를 반환하는데, 키는 사용자이름이고 값은 사용자가 Push할 수 있는 경로의 목록이다:

```
def get_acl_access_data(acl_file)
```

```
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

이 함수가 ACL 파일을 처리하고 나서 반환하는 결과는 아래와 같다:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

바로 사용할 수 있는 권한 정보를 만들었다. 이제 Push하는 파일을 그 사용자가 Push할 수 있는지 없는지 알아내야 한다.
`git log` 명령에 `--name-only` 옵션을 주면 해당 커밋에서 수정된 파일이 뭔지 알려준다. `git log` 명령은 2장에서 다루었다:

```
$ git log -1 --name-only --pretty=format:'' 9f585d

README
lib/test.rb
```

`get_acl_access_data` 메소드를 호출해서 ACL 정보를 구하고, 각 커밋에 들어 있는 파일 목록도 얻은 다음에, 사용자가 모든 커밋을 Push할 수 있는지 판단한다:

```
# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
```

```
    access = get_acl_access_data('acl')

    # see if anyone is trying to push something they can't
    new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
    new_commits.each do |rev|
      files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
      files_modified.each do |path|
        next if path.size == 0
        has_file_access = false
        access[$user].each do |access_path|
          if !access_path || # user has access to everything
            (path.index(access_path) == 0) # access to this path
            has_file_access = true
          end
        end
        if !has_file_access
          puts "[POLICY] You do not have access to push to #{path}"
          exit 1
        end
      end
    end
  end

  check_directory_perms
```

어렵지 않다. 먼저 `git rev-list` 명령으로 서버에 Push하려는 커밋이 무엇인지 알아낸다. 그리고 각 커밋에서 수정한 파일이 어떤 것들이 있는지 찾고, 해당 사용자가 모든 파일에 대한 권한이 있는지 확인한다. Rubyism 철학에 따르면 `path.index(access_path) == 0`이란 표현은 불명확하다. 이 표현은 해당 파일의 경로가 `access_path`로 시작할 때 참이라는 뜻이다. 그러니까 `access_path`가 단순히 허용된 파일 하나를 의미하는 것이 아니라 `access_path`로 시작하는 모든 파일을 의미한다.

이제 사용자는 메시지 규칙을 어겼거나 권한이 없는 파일이 포함된 커밋은 어떤 것도 Push하지 못한다.

Fast-Forward Push만 허용하기

이제 Fast-forward Push가 아니면 거절되게 해보자. `receive.denyDeletes`와 `receive.denyNonFastForwards` 설정으로 간단하게 거절할 수 있다. 하지만, 그 이전 버전에는 꼭 훅으로 구현해야 했다. 게다가 특정 사용자만 제한하거나 허용하려면 훅으로 구현해야 한다.

기존에 있던 커밋이 Push하는 브랜치에 없으면 Fast-forward Push가 아니라고 판단한다. 커밋 하나라도 없으면 거절하고 모두 있으면 Fast-forward Push이므로 그대로 둔다:

```
# enforces fast-forward only pushes
def check_fast_forward
```

```
  missed_refs = `git rev-list #{$newrev}..#{$oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

이 정책을 다 구현해서 update 스크립트에 넣고 `chmod u+x .git/hooks/update` 명령으로 실행 권한을 준다. 그리고 나서 `-f` 옵션을 주고 강제로 Push하면 아래와 같이 실패한다:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Cannot push a non-fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

정책과 관련해 하나씩 살펴보자. 먼저 훅이 실행될 때마다 다음 메시지가 출력된다.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

이것은 update 스크립트 맨 윗부분에서 표준출력(STDOUT)에 출력한 내용이다. 스크립트에서 표준출력으로 출력하면 클라이언트로 전송된다. 이점을 꼭 기억하자.
그리고 아래의 에러 메시지를 보자:

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

첫 번째 줄은 스크립트에서 직접 출력한 것이고 나머지 두 줄은 Git이 출력해 주는 것이다. 이 메시지는 update 스크립트에서 0이 아닌 값을 반환해서 Push할 수 없다는 메시지다. 그리고 마지막 메시지를 보자:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

이 메시지는 훅에서 거절된 것이라고 말해주는 것이고 브랜치가 거부될 때마다 하나씩 출력된다.
게다가 Push하는 커밋에 커밋 메시지 규칙을 지키지 않은 것이 하나라도 있으면 아래와 같은 에러 메시지를 보여준다:

```
[POLICY] Your message is not formatted correctly
```

그리고 누군가 권한이 없는 파일을 수정해서 Push해도 에러 메시지를 출력한다. 예를 들어 문서 담당자가 `lib` 디렉토리에 있는 파일을 수정해서 커밋하면 아래와 같은 메시지가 출력된다:

```
[POLICY] You do not have access to push to lib/test.rb
```

이제 서버 훅은 다 만들었다. 앞으로는 update 스크립트가 항상 실행될 것이기 때문에 저장소를 되돌릴 수 없고, 커밋 메시지도 규칙대로 작성해야 하고, 권한이 있는 파일만 Push할 수 있다.

7.4.2 클라이언트 훅

서버 훅의 단점은 Push할 때까지 Push할 수 있는지 없는지 알 수 없다는데 있다. 기껏 공들여 정성껏 구현했는데 막상 Push할 수 없으면 곤혹스럽다. 히스토리를 제대로 고치는 일은 정신건강에 매우 해롭다.
이 문제는 클라이언트 훅으로 해결한다. 클라이언트 훅으로 서버가 거부할지 말지 검사한다. 사람들은 커밋하기 전에, 그러니까 시간이 지나 고치기 어려워지기 전에 문제를 해결할 수 있다. Clone할 때 이 훅은 전송되지 않기 때문에 다른 방법으로 동료에게 배포해야 한다. 그 훅을 가져다 `.git/hooks` 디렉토리에 복사하고 실행할 수 있게 만든다. 이 훅 파일을 프로젝트에 넣어서 배포해도 되고 Git 훅 프로젝트를 만들어서 배포해도 된다. 하지만, 자동으로 설치하는 방법은 없다.
커밋 메시지부터 검사해보자. 이 훅이 있으면 커밋 메시지가 구리다고 서버가 뒤늦게 거절하지 않는다. 이것은 `commit-msg` 훅으로 구현한다. 이 훅은 커밋 메시지가 저장된 파일을 첫 번째 아규먼트로 입력받는다. 그 파일을 읽어 패턴을 검사한다. 필요한 패턴이 없으면 커밋을 중단시킨다:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)
```

```
$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

이 스크립트를 `.git/hooks/commit-msg`라는 파일로 만들고 실행권한을 준다. 커밋이 메시지 규칙을 어기면 아래와 같은 메시지를 보여 준다:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

커밋하지 못했다. 하지만, 커밋 메지시를 바르게 작성하면 커밋할 수 있다:

```
$ git commit -am 'test [ref: 132]'
[master e05c914] test [ref: 132]
 1 files changed, 1 insertions(+), 0 deletions(-)
```

그리고 아예 권한이 없는 파일을 수정 못하게 할 때는 `pre-commit` 훅을 이용한다. 사전에 `.git` 디렉토리 안에 ACL 파일을 가져다 놓고 아래와 같이 작성한다:

```
#!/usr/bin/env ruby

$user    = ENV['USER']

# [ insert acl_access_data method from above ]

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('.git/acl')

  files_modified = `git diff-index --cached --name-only HEAD`.split("\n")
  files_modified.each do |path|
    next if path.size == 0
    has_file_access = false
    access[$user].each do |access_path|
    if !access_path || (path.index(access_path) == 0)
      has_file_access = true
    end
```

```
    if !has_file_access
      puts "[POLICY] You do not have access to push to #{path}"
      exit 1
    end
  end
end

check_directory_perms
```

내용은 서버 훅과 똑같지만 두 가지가 다르다. 첫째, 클라이언트 훅은 Git 디렉토리가 아니라 워킹 디렉토리에서 실행하기 때문에 ACL 파일 위치가 다르다. 그래서 ACL 파일 경로를 수정해야 한다:

```
access = get_acl_access_data('acl')
```

이 부분을 아래와 같이 바꾼다:

```
access = get_acl_access_data('.git/acl')
```

두 번째 차이점은 파일 목록을 얻는 방법이다. 서버 훅에서는 커밋에 있는 파일을 모두 찾았지만 여기서는 아직 커밋하지도 않았다. 그래서 Staging Area의 파일 목록을 이용한다:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

이 부분을 아래와 같이 바꾼다:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

이 두 가지 점만 다르고 나머지는 똑같다. 보통은 리모트 저장소의 계정과 로컬의 계정도 같다. 다른 계정을 사용하려면 `$user` 환경변수에 누군지 알려야 한다.

Fast-forward Push인지 확인하는 일이 남았다. 보통은 Fast-forward가 아닌 Push는 좀 드물다. Fast-forward가 아닌 Push를 하려면 Rebase로 이미 Push한 커밋을 바꿔 버렸거나 전혀 다른 로컬 브랜치를 Push하는 경우다.

어쨌든 이 서버는 Fast-forward Push만 허용하기 때문에 이미 Push한 커밋을 수정했다면 그건 아마 실수로 그랬을 것이다. 이 실수를 막는 훅을 살펴보자.

아래는 이미 Push한 커밋을 Rebase하지 못하게 하는 pre-rebase 스크립트다. 이 스크립트는 먼저 Rebase할 커밋 목록을 구하고 커밋이 리모트 레퍼런스/브랜치에 들어 있는지 확인한다. 커밋이 한 개라도 리모트 레퍼런스/브랜치에 들어 있으면 Rebase할 수 없다:

```
#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { ¦r¦ r.strip }

target_shas.each do ¦sha¦
  remote_refs.each do ¦remote_ref¦
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end
```

이 스크립트는 6장 '리비전 조회하기' 절에서 설명하지 않은 표현을 사용했다. 아래의 표현은 이미 Push한 커밋 목록을 얻어오는 부분이다:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

`SHA^@`은 해당 커밋의 모든 부모를 가리킨다. 그러니까 이 명령은 지금 Push하려는 커밋에서 리모트 저장소의 커밋에 도달할 수 있는지 확인하는 명령이다. 즉, Fast-forward인지 확인하는 것이다.
이 방법은 매우 느리고 보통은 필요 없다. 어차피 Fast-forward가 아닌 Push은 `-f` 옵션을 주어야 Push할 수 있다. 문제가 될만한 Rebase를 방지할 수 있다는 것을 보여주려고 이 예제를 설명했다.

7.5 요약

Git을 프로젝트에 맞추는 방법을 배웠다. 주요한 서버/클라이언트 설정 방법, 파일 단위로 설정하는 Git Attributes, 이벤트 훅, 정책을 강제하는 방법을 배웠다. 이제 필요한 Workflow를 만들고 Git을 거기에 맞게 설정할 수 있을 것이다.

8장

Git으로 이전하기

Git은 완벽하지 않다. 프로젝트를 전부 Git으로 옮기기는 어렵다. 프로젝트가 특정 VCS 시스템에 매우 의존적으로 개발 됐을 수도 있다. 보통은 Subversion에 의존적이다. 이번 장은 `git svn`이라는 Git과 Subversion을 양방향으로 이어 주는 도구를 알아 보며 시작한다.

언젠가 이미 존재하는 프로젝트 환경을 Git으로 변경하고 싶게 될 것이다. 이 장의 나머지 부분에서 프로젝트를 Git으로 변경하는 방법에 대해 다룰 것이다. 먼저 Subversion에서 프로젝트를 옮겨 오는 방법을 설명하고 그 다음에는 Perforce, 그리고 스크립트를 직접 만들어서 잘 쓰지 않는 VCS에서도 프로젝트를 옮기는 방법을 다룰 것이다.

8.1 Git과 Subversion

현재도 많은 오픈소스 프로젝트와 수 많은 기업 프로젝트는 Subversion으로 소스코드를 관리한다. 10여년간 Subversion이 가장 인기있는 오픈소스 VCS 도구였다. Subversion은 그 이전 시대에서 가장 많이 사용하던 CVS와 많이 닮았다.

Git이 자랑하는 또 하나의 기능은 `git svn`이라는 양방향 Subversion 지원 도구이다. Git을 Subversion 클라이언트로 사용할 수 있기 때문에 로컬에서는 Git의 기능을 활용하고 Push 할 때는 Subversion 서버에 Push한다. 로컬 브랜치와 Merge, Staging Area, Rebase, Cherry-pick 등의 Git 기능을 충분히 사용할 수 있다. 같이 일하는 동료는 빛 한줄기 없는 선사시대 동굴에서 일하겠지만 말이다. `git svn`은 기업에서 git을 사용할 수 있도록 돕는 출발점이다. 우리가 Git을 도입하기 위해 기업내에서 노력하는 동안 동료가 효율적으로 환경을 바꿀 수 있도록 도움을 줄 수 잇다. Subversion 지원 도구는 우리를 DVCS 세상으로 인도하는 붉은 알약과 같은 것이다.

8.1.1 git svn

Git과 Subversion을 이어주는 명령은 `git svn` 으로 시작한다. 이 명령 뒤에 추가하는 명령이 몇 가지 더 있으며 간단한 예제를 보여주고 설명한다.

`git svn` 명령을 사용할 때는 절름발이인 Subversion을 사용하고 있다는 점을 염두하자. 우리가 로컬 브랜치와 Merge를 맘대로 쓸 수 있다고 하더라도 최대한 일직선으로 히스토리를 유지하는것이 좋다. Git 저장소처럼 사용하지 않는다.

히스토리를 재작성해서 Push하지 말아야 한다. Git을 사용하는 동료들끼리 따로 Git 저장소에 Push 하지도 말아야 한다. Subversion은 단순하게 일직선 히스토리만 가능하다. 팀원중 일부는 SVN을 사용하고 일부는 Git을 사용하는 팀이라면 SVN Server를 사용해서 협업하는 것이 좋다. 그래야 삶이 편해진다.

8.1.2 설정하기

git svn을 사용하려면 SVN 저장소가 하나 필요하다. 저장소에 쓰기 권한이 있어야 한다. 필자의 test 저장소를 복사한다. Subversion(1.4 이상)에 포함된 svnsync라는 도구를 사용하여 SVN 저장소를 복사한다. 테스트용 저장소가 필요해서 Google Code에 새로 Subversion 저장소를 하나 만들었다. protobuf 라는 프로젝트의 일부 코드를 복사했다. protobuf는 네트워크 전송에 필요한 구조화된 데이터 (프로토콜 같은 것들)의 인코딩을 도와주는 도구이다.

로컬 Subversion 저장소를 하나 만든다:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

그리고 모든 사용자가 revprops 속성을 변경할 수 있도록 항상 0을 반환하는 pre-revprop-change 스크립트를 준비한다(역주: 파일이 없거나, 다른 이름으로 되어있을 수 있다. 이 경우 아래 내용으로 새로 파일을 만들고 실행 권한을 준다):

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

이제 svnsync init 명령으로 다른 Subversion 저장소를 로컬로 복사할 수 있도록 지정한다:

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

이렇게 다른 저장소의 주소를 설정하면 복사할 준비가 된다. 아래 명령으로 저장소를 실제로 복사한다:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

이 명령은 몇 분 걸리지 않는다. 저장하는 위치가 로컬이 아니라 리모트 서버라면 오래 걸린다. 커밋이 100개 이하라고 해도 오래 걸린다. Subversion은 한번에 커밋을 하나씩 받아서 Push하기 때문에 엄청나게 비효율적이다. 하지만, 저장소를 복사하는 다른 방법은 없다.

8.1.3 시작하기

이제 갖고 놀 Subversion 저장소를 하나 준비했다. `git svn clone` 명령으로 Subversion 저장소 전체를 Git 저장소로 가져온다. 만약 Subversion 저장소가 로컬에 있는 것이 아니라 리모트 서버에 있으면 `file:///tmp/test-svn` 부분에 서버 저장소의 URL을 적어 준다.

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git/
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
      A    m4/acx_pthread.m4
      A    m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
 file:///tmp/test-svn/branches/my-calc-branch r76
```

이 명령은 사실 SVN 저장소 주소를 주고 `git svn init`과 `git svn fetch` 명령을 순서대로 실행한 것과 같다. 이 명령은 시간이 좀 걸린다. 테스트용 프로젝트는 커밋이 75개 정도 밖에 안되서 시간이 많이 걸리지 않는다. Git은 커밋을 한번에 하나씩 일일이 기록해야 한다. 커밋이 수천개인 프로젝트라면 몇 시간 혹은 몇 일이 걸릴 수도 있다.
`-T trunk -b branches -t tags` 부분은 Subversion이 어떤 브랜치 구조를 가지고 있는지 Git에게 알려주는 부분이다. Subversion 표준 형식과 다르면 이 옵션 부분에서 알맞은 이름을 지정해준다. 표준 형식을 사용한다면 간단하게 `-s` 옵션을 사용한다. 즉 아래의 명령도 같은 의미이다.

```
$ git svn clone file:///tmp/test-svn -s
```

Git에서 브랜치와 태그 정보가 제대로 보이는 것을 확인한다:

```
$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk
```

git svn 도구가 리모트 브랜치의 이름을 어떻게 짓는지 알아야 한다. Git 저장소를 Clone할 때는 보통 origin/[branch]처럼 리모트 저장소 이름이 들어간 브랜치 이름으로 만들어진다. git svn은 우리가 리모트 저장소를 딱 하나만 사용한다고 가정한다. 그래서 리모트 저장소의 이름을 붙여서 브랜치를 관리하지 않는다. Plumbing 명령어인 show-ref 명령으로 리모트 브랜치의 정확한 이름을 확인할 수 있다.

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

일반적인 Git 저장소라면 아래와 비슷하다:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

이 결과를 보면 리모트 저장소가 두 개 있다. gitserver라는 리모트 저장소에 master 브랜치가 있고 origin이라는 리모트 저장소에 master, testing 브랜치가 있다.
git svn으로 저장소를 가져오면 Subversion 태그는 Git 태그가 아니라 리모트 브랜치로 등록되는 점을 잘 기억하자. git svn은 Subversion 태그를 tags라는 리모트 서버에 있는 브랜치처럼 만든다.

8.1.4 Subversion 서버에 커밋하기

자 작업할 Git 저장소는 준비했다. 무엇인가 수정하고 서버로 고친 내용을 Push해야 할 때가 왔다. Git을 Subversion의 클라이언트로 사용해서 수정한 내용을 전송한다. 어떤 파일을 수정하고 커밋을 하면 그 수정한 내용은 Git의 로컬 저장소에 저장된다. Subversion 서버에는 아직 반영되지 않는다.

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
 1 files changed, 1 insertions(+), 1 deletions(-)
```

이제 수정한 내용을 서버로 전송한다. Git 저장소에 여러개의 커밋을 쌓아놓고 한번에 Subversion 서버로 보낸다는 점을 잘 살펴보자. git svn dcommit 명령으로 서버에 Push한다.

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r79
       M      README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

이 명령은 새로 추가한 커밋을 모두 Subversion에 커밋하고 로컬 Git 커밋을 다시 만든다. 커밋을 다시 만들기 때문에 이미 저장된 커밋의 SHA-1 체크섬이 바뀐다. 그래서 리모트 Git 저장소와 Subversion 저장소를 함께 사용하면 안된다. 새로 만들어진 커밋을 살펴보면 아래와 같이 `git-svn-id`가 추가된다:

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sat May 2 22:06:44 2009 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

원래 `97031e5`로 시작하는 SHA 체크섬이 지금은 `938b1a5`로 시작한다. 만약 Git 서버와 Subversion 서버에 함께 Push하고 싶으면 우선 Subversion 서버에 `dcommit`으로 Push를 하고 그 다음에 Git 서버에 Push해야 한다.

8.1.5 새로운 변경사항 받아오기

다른 개발자와 함께 일하는 과정에서 다른 개발자가 Push한 상태에서 Push를 하면 충돌이 날 수 있다. 충돌을 해결하지 않으면 서버로 Push할 수 없다. 충돌이 나면 `git svn` 명령은 아래와 같이 보여준다:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

이런 상황에서는 `git svn rebase` 명령으로 이 문제를 해결한다. 이 명령은 변경사항을 서버에서 내려받고 그 다음에 로컬의 변경사항을 그 위에 적용한다:

```
$ git svn rebase
       M      README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

그러면 서버 코드 위에 변경사항을 적용하기 때문에 성공적으로 `dcommit` 명령을 마칠 수 있다:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      README.txt
Committed r81
       M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Push하기 전에 서버의 내용을 Merge하는 Git과 달리 `git svn`은 충돌이 날때에만 서버에 업데이트할 것이 있다고 알려 준다. 이 점을 꼭 기억해야 한다. 만약 다른 사람이 한 파일을 수정하고 내가 그 사람과 다른 파일을 수정한다면 `dcommit`은 성공적으로 수행된다:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      configure.ac
Committed r84
       M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
       M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48fce3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M   autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Push하고 나면 프로젝트 상태가 달라진다는 점을 기억해야 한다. 충돌이 없으면 변경사항이 바램대로 적용되지 않아도 알려주지 않는다. 이 부분이 Git과 다른 점이다. Git에서는 서버로 보내기 전에 프로젝트 상태를 전부 테스트할 수 있다. SVN은 서버로 커밋하기 전과 후의 상태가 동일하다는 것이 보장되지 않는다.

`git svn rebase` 명령으로도 Subversion 서버의 변경사항을 가져올 수 있다. 커밋을 보낼 준비가 안됐어도 괜찮다. `git svn fetch` 명령을 사용해도 되지만 `git svn rebase` 명령은 변경사항을 가져오고 적용까지 한 번에 해준다.

```
$ git svn rebase
       M      generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

수시로 `git svn rebase` 명령을 사용하면 로컬 코드를 항상 최신 버전으로 유지할 수 있다. 이 명령을 사용하기 전에 워킹 디렉토리를 깨끗하게 만드는 것이 좋다. 깨끗하지 못하면 Stash를 하거나 임시로 커밋하고 나서 `git svn rebase` 명령을 실행하는 것이 좋다. 깨끗하지 않으면 충돌이 나서 Rebase가 중지될 수 있다.

8.1.6 Git 브랜치 문제

Git에 익숙한 사람이면 일을 할 때 먼저 토픽 브랜치를 만들고, 일을 끝낸 다음에, Merge하는 방식을 쓰려고 할 것이다. 하지만, `git svn`으로 Subversion 서버에 Push할 때에는 브랜치를 Merge하지 않고 Rebase해야 한다. Subversion은 일직선 히스토리 밖에 모르고 Git의 Merge도 알지 못한다. 그래서 `git svn`은 첫 번째 부모 정보만 사용해서 Git 커밋을 Subversion 커밋으로 변경한다.
예제를 하나 살펴보자. `experiment` 브랜치를 하나 만들고 2개의 변경사항을 커밋한다. 그리고 `master` 브랜치로 Merge하고 나서 `dcommit` 명령을 수행하면 아래와 같은 모양이 된다:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
       M      CHANGES.txt
Committed r85
       M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
       M      COPYING.txt
       M      INSTALL.txt
Committed r86
       M      INSTALL.txt
       M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Merge 커밋이 들어 있는 히스토리에서 `dcommit` 명령을 실행한다. 그리고 나서 Git 히스토리를 살펴보면 `experiment` 브랜치의 커밋은 재작성되지 않았다. 대신 Merge 커밋만 SVN 서버로 전송됐을 뿐이다. 누군가 이 것을 내려 받으면 결과가 합쳐진 Merge 커밋 하나만 볼 수 있다. 다른 사람은 언제 어디서 커밋한 것인지 알 수 없다.

8.1.7 Subversion의 브랜치

Subversion의 브랜치는 Git의 브랜치와 달라서 가능한 사용을 하지 않는 것이 좋다. 하지만 `git svn`으로도 Subversion 브랜치를 관리할 수 있다.

SVN 브랜치 만들기

Subversion 브랜치를 만들려면 `git svn branch [branchname]` 명령을 사용한다:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

이 명령은 Subversion의 `svn copy trunk branches/opera` 명령과 동일하다. 이 명령은 브랜치를 Checkout해주지 않는다는 것을 주의해야 한다. 여기서 커밋하면 `opera` 브랜치가 아니라 `trunk` 브랜치에 커밋된다.

8.1.8 Subversion 브랜치 넘나들기

`dcommit` 명령은 어떻게 커밋 할 브랜치를 결정할까? Git은 히스토리에 있는 커밋중에서 가장 마지막으로 기록된 Subversion 브랜치를 찾는다. 즉, 현 브랜치 히스토리의 커밋 메시지에 있는 `git-svn-id` 항목을 읽는 것이기 때문에 오직 한 브랜치에만 전송할 수 있다.

동시에 여러 브랜치에서 작업하려면 Subversion 브랜치에 `dcommit`할 수 있는 로컬 브랜치가 필요하다. 이 브랜치는 Subversion 커밋에서 시작하는 브랜치다. 아래와 같이 `opera` 브랜치를 만들면 독립적으로 일 할 수 있다:

```
$ git branch opera remotes/opera
```

`git merge` 명령으로 `opera` 브랜치를 `trunk` 브랜치(`master` 브랜치 역할)에 Merge한다. 하지만 `-m` 옵션을 주고 적절한 커밋 메시지를 작성하지 않으면 아무짝에 쓸모없는 "Merge branch opera" 같은 메시지가 커밋된다.

`git merge` 명령으로 Merge한다는 것에 주목하자. Git은 자동으로 공통 커밋을 찾아서 Merge에 참고하기 때문에 Subversion에서 하는 것보다 Merge가 더 잘된다. 여기서 생성되는 Merge 커밋은 일반적인 Merge 커밋과 다르다. 이 커밋을 Subversion 서버에 Push해야 하지만 Subversion에서는 부

모가 2개인 커밋이 있을 수 없다. 그래서 Push하면 브랜치에서 만들었던 커밋 여러개가 하나로 합쳐진(squash된) 것처럼 Push된다. 그래서 일단 Merge하면 취소하거나 해당 브랜치에서 계속 작업하기 어렵다. `dcommit` 명령을 수행하면 Merge한 브랜치의 정보를 어쩔 수 없이 잃어버리게 된다. Merge Base도 찾을 수 없게 된다. `dcommit` 명령은 Merge한 것을 `git merge --squash`로 Merge한 것과 똑 같이 만들어 버린다. Branch를 Merge한 정보는 저장되지 않기 때문에 이 문제를 해결할 방법이 없다. 문제를 최소화하려면 trunk에 Merge하자마자 해당 브랜치를(여기서는 `opera`) 삭제하는 것이 좋다.

8.1.9 Subversion 명령

`git svn` 명령은 Git으로 전향하기 쉽도록 Subversion에 있는 것과 비슷한 명령어를 지원한다. 아마 여기서 설명하는 명령은 익숙할 것이다.

SVN 형식의 히스토리

Subversion에 익숙한 사람은 Git 히스토리를 SVN 형식으로 보고 싶을 수도 있다. `git svn log` 명령은 SVN 형식으로 히스토리를 보여준다:

```
$ git svn log
------------------------------------------------------------------------
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines

autogen change

------------------------------------------------------------------------
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines

Merge branch 'experiment'

------------------------------------------------------------------------
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines

updated the changelog
```

`git svn log`명령에서 기억해야 할 것은 두 가지다. 우선 오프라인에서 동작한다는 점이다. SVN의 `svn log` 명령어는 히스토리 데이터를 조회할 때 서버가 필요하다. 둘째로 이미 서버로 전송한 커밋만 출력해준다. 아직 `dcommit` 명령으로 서버에 전송하지 않은 로컬 Git 커밋은 보여주지 않는다. Subversion 서버에는 있지만 아직 내려받지 않은 변경사항도 보여주지 않는다. 즉, 현재 알고있는 Subversion 서버의 상태만 보여준다.

SVN 어노테이션

`git svn log` 명령이 `svn log` 명령을 흉내내는 것처럼 `git svn blame [FILE]` 명령으로 `svn annotate` 명령을 흉내낼 수 있다. 실행한 결과는 아래와 같다:

```
$ git svn blame README.txt
 2   temporal Protocol Buffers - Google's data interchange format
 2   temporal Copyright 2008 Google Inc.
 2   temporal http://code.google.com/apis/protocolbuffers/
 2   temporal
22   temporal C++ Installation - Unix
22   temporal =======================
 2   temporal
79    schacon Committing in git-svn.
78    schacon
 2   temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2   temporal Buffer compiler (protoc) execute the following:
 2   temporal
```

다시 한번 말하지만 이 명령도 아직 서버로 전송하지 않은 커밋은 보여주지 않는다.

SVN 서버 정보

`svn info` 명령은 `git svn info` 명령으로 대신할 수 있다:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

`blame`이나 `log`명령이 오프라인으로 동작하듯이 이 명령도 오프라인으로 동작한다. 서버에서 가장 최근에 내려받은 정보를 출력한다.

Subversion에서 무시하는것 무시하기

Subversion 저장소를 클론하면 쓸데 없는 파일을 커밋하지 않도록 `svn:ignore` 속성을 `.gitignore` 파일로 만들고 싶을 것이다. `git svn`에는 이 문제와 관련된 명령이 두 가지 있다. 하나는 `git svn create-ignore` 명령이다. 해당 위치에 커밋할 수 있는 `.gitignore` 파일을 생성해준다.
두 번째 방법은 `git svn show-ignore` 명령이다. `.gitignore`에 추가할 목록을 출력해 준다. 프로젝트의 exclude 파일로 결과를 리다이렉트할 수 있다:

```
$ git svn show-ignore > .git/info/exclude
```

이렇게 하면 `.gitignore` 파일로 프로젝트를 더럽히지 않아도 된다. 혼자서만 Git을 사용하는 거라면 다른 팀원들은 프로젝트에 `.gitignore` 파일이 있는 것을 싫어 할 수 있다.

8.1.10 Git-Svn 요약

`git svn` 도구는 여러가지 이유로 Subversion 서버를 사용해야만 하는 상황에서 빛을 발한다. 하지만 Git의 모든 장점을 이용할 수는 없다. Git과 Subversion은 다르기 때문에 혼란이 빚어질 수도 있다. 이런 문제에 빠지지 않기 위해서 다음 가이드라인을 지켜야 한다:

- Git 히스토리를 일직선으로 유지하라. `git merge`로 Merge 커밋이 생기지 않도록 하라. Merge 말고 Rebase로 변경사항을 Master 브랜치에 적용하라.
- 따로 Git 저장소 서버를 두지 말라. 클론을 빨리 하기 위해서 잠깐 하나 만들어 쓰는 것은 무방하나 절대로 Git 서버에 Push하지는 말아야 한다. `pre-receive` 훅에서 `git-svn-id`가 들어 있는 커밋 메시지는 거절하는 방법도 괜찮다.

이러한 가이드라인을 잘 지키면 Subversion 서버도 쓸만하다. 그래도 Git 서버를 사용할 수 있으면 Git 서버를 사용하는 것이 훨씬 좋다.

8.2 Git으로 옮기기

다른 VCS를 사용하는 프로젝트를 Git으로 옮기고 싶다면 우선 프로젝트를 Git으로 이전(Migrate)해야 한다. 이번 절에서는 Git에 들어 있는 Importer를 살펴보고 직접 Importer를 만드는 방법을 알아본다.

8.2.1 가져오기

많이 사용하는 Subversion과 Perforce 프로젝트를 이전하는 방법을 살펴보자. 이 두 VCS에서 Git으로 이전하고자 하는 사람이 많고 Importer도 이미 Git에 들어 있다.

8.2.2 Subversion

`git svn`을 설명하는 절을 읽었으면 쉽게 `git svn clone` 명령으로 저장소를 가져올 수 있다. 가져오고 나서 Subversion 서버는 중지하고 Git 서버를 만들고 사용하면 된다. 만약 히스토리 정보가 필요하면 (느린) Subversion 서버 없이 로컬에서 조회할 수 있다.

우선 가져오기에 시간이 많이 드니까 일단 가져오기를 시작하자. 이 가져오기 기능에 문제가 좀 있다. 첫 번째 문제는 Author 정보이다. Subversion에서는 커밋하려면 해당 시스템 계정이 있어야 한다. `blame`이나 `git svn log` 같은 명령에서 `schacon`이라는 이름을 봤을 것이다. 이 정보를 Git 형식의 정보로 변경하려면 Subversion 사용자와 Git Author를 연결시켜줘야 한다. Subversion 사용자 이름과 Git Author 간에 연결을 해줘서 이 Author 정보를 Git 스타일의 Author 정보로 변경한다. `users.txt`라는 파일을 아래와 같이 만든다:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

SVN에 기록된 Author 이름을 아래 명령으로 조회한다:

```
$ svn log ^/ --xml | grep -P "^<author" | sort -u | \
      perl -pe 's/<author>(.*?)<\/author>/$1 = /' > users.txt
```

우선 XML 형식으로 SVN 로그를 출력하고, 거기서 Author 정보만 찾고, 중복된 것을 제거하고, XML 태그는 버린다. 물론 `grep`, `sort`, `perl` 명령이 동작하는 시스템에서만 이 명령을 사용할 수 있다. 이 결과에 Git Author 정보를 더해서 `users.txt`를 만든다.
이 파일을 `git svn` 명령에 전달하면 보다 정확한 Author 정보를 Git 저장소에 남길 수 있다. 그리고 `git svn`의 `clone`이나 `init` 명령에 `--no-metadata` 옵션을 주면 Subversion의 메타데이터를 저장하지 않는다. 해당 명령은 아래와 같다:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
      --authors-file=users.txt --no-metadata -s my_project
```

`my_project` 디렉토리에 진짜 Git 저장소가 생성된다. 결과는 아래와 같지 않고:

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk

    git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-
    be05-5f7a86268029
```

아래와 같다:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Author 정보가 훨씬 Git답고 `git-svn-id` 항목도 기록되지 않았다.

이제 뒷 정리를 해야 한다. `git svn`이 만들어 준 이상한 브랜치나 태그를 제거한다. 우선 이상한 리모트 태그를 모두 진짜 Git 태그로 옮긴다. 그리고 리모트 브랜치도 로컬 브랜치로 옮긴다.
아래와 같이 태그를 진정한 Git 태그로 만든다:

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -v @ | while read tagname; do git tag "$tagname" "tags/$tagname"; git branch -r -d "tags/$tagname"; done
```

`tags/` 로 시작하는 리모트 브랜치를 가져다 (Lightweight) 태그로 만들었다.
`refs/remotes` 밑에 있는 레퍼런스는 전부 로컬 브랜치로 만든다:

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -v @ | while read branchname; do git branch "$branchname" "refs/remotes/$branchname"; git branch -r -d "$branchname"; done
```

이제 모든 태그와 브랜치는 진짜 Git 태그와 브랜치가 됐다. Git 서버를 새로 추가를 하고 지금까지 작업한 것을 Push하는 일이 남았다. 아래처럼 리모트 서버를 추가한다:

```
$ git remote add origin git@my-git-server:myrepository.git
```

분명 모든 브랜치와 태그를 Push하고 싶을 것이다:

```
$ git push origin --all
```

모든 브랜치와 태그를 Git 서버로 깔끔하게 잘 옮겼다.

8.2.3 Perforce

이제 Perforce 차례다. Preforce Importer도 Git에 들어 있다. Git 1.7.11 이전 버전을 사용한다면 소스코드의 `contrib` 에 포함되어 있다.이런 경우라면 Perforce Importer를 사용하기 위해 우선 git.kernel.org에서 Git 소스코드를 가져와야 한다:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/contrib/fast-import
```

`git-p4` 라는 Python 스크립트는 `fast-import` 디렉토리에 있다. 그리고 Python과 `p4`가 설치돼 있어야 이 스크립트가 동작한다. Perforce Public Depot에 있는 Jam 프로젝트를 Git으로 옮겨보자. 우선 Perfoce Depot의 주소를 P4PORT 환경변수에 설정한다:

```
$ export P4PORT=public.perforce.com:1666
```

`git-p4 clone` 명령으로 Perforce 서버에서 Jam 프로젝트를 가져온다. 이 명령에 Depot, 프로젝트 경로, 프로젝트를 가져올 경로를 주면 된다:

```
$ git-p4 clone //public/jam/src@all /opt/p4import
Importing from //public/jam/src@all into /opt/p4import
Reinitialized existing Git repository in /opt/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 4409 (100%)
```

`/opt/p4import` 디렉토리로 이동해서 `git log` 명령을 실행하면 프로젝트 정보를 볼 수 있다:

```
$ git log -2
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

    [git-p4: depot-paths = "//public/jam/src/": change = 4409]

commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c

    [git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

커밋마다 `git-p4` 라는 ID 항목이 들어가 있다. 나중에 Perforce Change Number가 필요해질 수도 있으니 커밋에 그대로 유지하는 편이 좋다. 하지만 ID를 지우고자 한다면 지금 하는 것이 가장 좋다. `git filter-branch` 명령으로 한방에 삭제한다:

```
$ git filter-branch --msg-filter '
        sed -e "/^\[git-p4:/d"
```

```
'
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

`git log` 명령을 실행하면 모든 SHA-1 체크섬이 변경됐고 커밋 메시지에서 `git-p4` 항목도 삭제된 것을 확인할 수 있다.

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date:   Thu Aug 19 10:18:45 2004 -0800

    Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
    the main part of the document.  Built new tar/zip balls.

    Only 16 months later.

commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date:   Tue Apr 22 20:51:34 2003 -0800

    Update derived jamgram.c
```

이제 새 Git 서버에 Push하면 된다.

8.2.4 직접 Importer 만들기

사용하는 VCS가 Subversion이나 Perforce가 아니면 인터넷에서 적당한 Importer를 찾아봐야 한다. CVS, Clear Case, Visual Source Safe 같은 시스템용 Importer가 좋은게 많다. 심지어 단순히 디렉토리 아카이브용 Importer에도 좋은게 있다. 사람들이 잘 안쓰는 시스템을 사용하고 있는데 적당한 Importer를 못 찾았거나 부족해서 좀 더 고쳐야 한다면 `git fast-import`를 사용한다. 이 명령은 표준입력으로 데이터를 입력받는데, 9장에서 배우는 저수준 명령어와 내부 객체를 직접 다루는 것보다 훨씬 쉽다. 먼저 사용하는 VCS에서 필요한 정보를 수집해서 표준출력으로 출력하는 스크립트를 만든다. 그리고 그 결과를 `git fast-import`의 표준입력으로 보낸다(pipe).

간단한 Importer를 작성해보자. `back_YYYY_MM_DD` 라는 디렉토리에 백업하면서 프로젝트를 진행하는 예제를 보자. Importer를 만들 때 디렉토리 상태는 아래와 같다:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
```

```
current
```

Importer를 만들기 전에 우선 Git이 어떻게 데이터를 저장하는지 알아야 한다. 이미 알고 있듯이 Git은 기본적으로 스냅샷을 가리키는 커밋 개체가 연결된 리스트이다. 스냅샷이 뭐고, 그걸 가리키는 커밋은 또 뭐고, 그 커밋의 순서가 어떻게 되는지 `fast-import`에 알려 줘야 한다. 이 것이 해야할 일의 전부다. 그러면 디렉토리마다 스냅샷을 만들고, 그 스냅샷을 가리키는 커밋 개체를 만들고, 이전 커밋과 연결 시킨다.

7장의 "정책 구현하기" 절에서 했던 것 처럼 Ruby로 스크립트를 작성한다. 필자는 Ruby를 많이 사용하기도 하고 Ruby가 읽기도 쉽다. 하지만 자신에게 익숙한 것을 사용해서 표준출력으로 적절한 정보만 출력할 수 있으면 된다. 그리고 윈도에서는 줄바꿈 문자에 CR(Carriage Return) 문자가 들어가지 않도록 주의해야 한다. `git fast-import` 명령은 윈도에서도 줄바꿈 문자로 CRLF 문자가 아니라 LF(Line Feed) 문자만 허용한다.

우선 해당 디렉토리로 이동해서 어떤 디렉토리가 있는지 살펴본다. 하위 디렉토리마다 스냅샷 하나가 되고 커밋 하나가 된다. 하위 디렉토리를 이동하면서 필요한 정보를 출력한다. 기본적인 로직은 아래와 같다:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

각 디렉토리에서 `print_export`를 호출하는데 이 함수는 아규먼트로 디렉토리와 이전 스냅샷 Mark를 전달받고 현 스냅샷 Mark를 반환한다. 그래서 적절히 연결 시킬 수 있다. `fast-import`에서 "Mark"는 커밋의 식별자를 말한다. 커밋을 하나 만들면 Mark도 같이 만들어 이 Mark로 다른 커밋과 연결 시킨다. 그래서 `print_export`에서 우선 해야 하는 일은 각 디렉토리 이름으로 Mark를 생성하는 것이다:

```
mark = convert_dir_to_mark(dir)
```

Mark는 정수 값을 사용해야 하기 때문에 디렉토리를 배열에 담고 그 인덱스를 Mark로 사용한다. 아래와 같이 작성한다:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

각 커밋을 가리키는 정수 Mark를 만들었고 다음은 커밋 메타데이터에 넣을 날짜 정보가 필요하다. 이 날짜는 디렉토리 이름에 있는 것을 가져다 사용한다. `print_export`의 두 번째 줄은 아래와 같다:

```
date = convert_dir_to_date(dir)
```

`convert_dir_to_date`는 아래와 같이 정의한다:

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

시간는 정수 형태로 반환한다. 마지막으로 메타정보에 필요한 것은 Author인데 이 것은 전역 변수 하나로 설정해서 사용한다:

```
$author = 'Scott Chacon <schacon@example.com>'
```

이제 Importer에서 출력할 커밋 데이터는 다 준비했다. 이제 출력해보자. 사용할 브랜치, 해당 커밋과 관련된 Mark, 커미터 정보, 커밋 메시지, 이전 커밋을 출력한다. 코드로 만들면 아래와 같다:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

우선 시간대(-0700) 정보는 편의상 하드코딩으로 처리했다. 각자의 시간대에 맞는 오프셋을 설정해야 한다. 커밋 메시지는 아래와 같은 형식을 따라야 한다:

```
data (size)\n(contents)
```

이 형식은 'data'라는 단어, 읽을 데이터의 크기, 줄바꿈 문자, 실 데이터로 구성된다. 이 형식을 여러 곳에서 사용해야 하므로 `export_data`라는 메소드로 만들어 놓는게 좋다:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

이제 남은 것은 스냅샷에 파일 내용를 포함시키는 것 뿐이다. 디렉토리로 구분돼 있기 때문에 어렵지 않다. 우선 `deleteall`이라는 명령을 출력하고 그 뒤에 모든 파일의 내용을 출력한다. 그런면 Git은 스냅샷을 잘 저장한다:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

중요: 대부분의 VCS는 리비전을 커밋간의 변화로 생각하기 때문에 `fast-import`에 추가/삭제/변경된 부분만 입력할 수도 있다. 스냅샷 사이의 차이를 구해서 `fast-import`에 넘길 수도 있지만 훨씬 복잡하다. 줄 수 있는 데이터는 전부 Git에 줘서 Git이 계산하게 해야 한다. 꼭 이렇게 해야 한다면 어떻게 데이터를 전달해야 하는지 `fast-import`의 ManPage를 참고하라.
파일 정보와 내용은 아래와 같은 형식으로 출력한다:

```
M 644 inline path/to/file
data (size)
(file contents)
```

644는 파일의 모드를 나타낸다(실행파일이라면 755로 지정해줘야 한다). `inline`은 다음 줄 부터는 파일 내용이라는 말하는 것이다. `inline_data` 메소드는 아래와 같다:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

파일 내용은 커밋 메시지랑 같은 방법을 사용하기 때문에 앞서 만들어 놓은 `export_data` 메소드를 다시 이용한다.
마지막으로 다음 커밋에 사용할 현 Mark 값을 반환한다:

```
return mark
```

중요: 윈도에서 실행할 때는 추가 작업이 하나 더 필요하다. 앞에서 얘기했지만 윈도는 CRLF를 사용하지만 `git fast-import`는 LF를 사용한다. 이 문제를 해결 하려면 Ruby가 CRLF 대신 LF를 사용하도록 알려 줘야 한다:

```
$stdout.binmode
```

모든게 끝났다. 스크립트를 실행하면 아래와 같이 출력된다:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

디렉토리를 하나 만들고 `git init` 명령을 실행해서 옮길 Git 프로젝트를 만든다. 그리고 그 프로젝트 디렉토리로 이동해서 이 명령의 표준출력을 `git fast-import` 명령의 표준입력으로 연결한다(pipe).

```
$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
---------------------------------------------------------------------
Alloc'd objects:       5000
Total objects:           18 (         1 duplicates                  )
      blobs  :            7 (         1 duplicates          0 deltas)
      trees  :            6 (         0 duplicates          1 deltas)
      commits:            5 (         0 duplicates          0 deltas)
      tags   :            0 (         0 duplicates          0 deltas)
Total branches:           1 (         1 loads     )
      marks:           1024 (         5 unique    )
      atoms:              3
Memory total:          2255 KiB
       pools:          2098 KiB
     objects:           156 KiB
---------------------------------------------------------------------
pack_report: getpagesize()            =       4096
pack_report: core.packedGitWindowSize =   33554432
pack_report: core.packedGitLimit      =  268435456
pack_report: pack_used_ctr            =          9
pack_report: pack_mmap_calls          =          5
pack_report: pack_open_windows        =          1 /          1
pack_report: pack_mapped              =       1356 /       1356
---------------------------------------------------------------------
```

성공적으로 끝나면 여기서 보여주는 것처럼 어떻게 됐는지 통계를 보여준다. 이 경우엔 브랜치 1개와 커밋 5개 그리고 개체 18개가 임포트됐다. `git log` 명령으로 히스토리 조회가 가능하다:

```
$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03
```

이 시점에서는 아무것도 Checkout하지 않았기 때문에 워킹 디렉토리에 아직 아무 파일도 없다. `master` 브랜치로 Reset해서 파일을 Checkout한다:

```
$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls
file.rb  lib
```

`fast-import` 명령으로 많은 일을 할 수 있다. 모드를 설정하고, 바이너리 데이터를 다루고, 브랜치를 여러 개 다루고, Merge하고, 태그를 달고, 진행상황을 보여 주고, 등등 무수히 많은 일을 할 수 있다. Git 소스의 `contrib/fast-import` 디렉토리에 복잡한 상황을 다루는 예제가 많다. 그 중 여기서 설명한 `git-p4` 스크립트가 좋은 예제이다.

8.3 요약

Subversion 프로젝트에서 Git을 사용하거나, 다른 VCS 저장소를 Git 저장소로 손실 없이 옮기는 방법에 대해 알아 봤다. 다음장에서는 Git 내부를 까본다. 필요하다면 바이트 하나하나 다루는 것도 가능하다.

9장

Git의 내부

여기까지 다 읽고 왔든 바로 9장부터 보기 시작했든지 간에 이제 마지막 장이다. 9장은 Git이 어떻게 구현돼 있고 내부적으로 어떻게 동작하는지 설명한다. Git이 얼마나 유용하고 강력한지 이해하려면 9장의 내용을 꼭 알아야 한다. 9장은 초보자에게 너무 혼란스럽고 불필요한 내용이라고 이야기하는 사람들도 있다. 그래서 필자는 본 내용을 책의 가장 마지막에 두었고 독자가 스스로 먼저 볼지 나중에 볼지 선택할 수 있도록 했다.

자 이제 본격적으로 살펴보자. 우선 Git은 기본적으로 Content-addressable 파일 시스템이고 그 위에 VCS 사용자 인터페이스가 있는 구조다. 뭔가 깔끔한 정의는 아니지만, 이 말이 무슨 의미인지는 차차 알게 된다.

Git 초기에는 (1.5 이전 버전) 사용자 인터페이스가 훨씬 복잡했었다. VCS가 아니라 파일 시스템을 강조했기 때문이었다. 최근 몇 년간 Git은 다른 VCS처럼 쉽고 간결하게 사용자 인터페이스를 다듬어 왔다. 하지만, 여전히 복잡하고 배우기 어렵다는 선입견이 있다.

우선 Content-addressable 파일 시스템은 정말 대단한 것이므로 먼저 다룬다. 그리고 나서 데이터 전송 원리를 배우고 마지막에는 저장소를 관리하는 법까지 배운다.

9.1 Plumbing 명령과 Porcelain 명령

이 책에서는 `checkout`, `branch`, `remote` 같은 30여 가지의 Git 명령을 사용했다. Git은 사실 사용자 친화적인 VCS이기 보다는 VCS로도 사용할 수 있는 툴킷이다. 저수준 명령어가 매우 많아서 저수준의 일도 쉽게 처리할 수 있다. 명령어 여러 개를 Unix 스타일로 함께 엮어서 실행하거나 스크립트에서 호출할 수 있도록 설계됐다. 이러한 저수준의 명령어는 "Plumbing" 명령어라고 부르고 좀 더 사용자에게 친숙한 사용자용 명령어는 "Porcelain" 명령어라고 부른다.

이 책의 앞 여덟 장은 Porcelain 명령어만 사용했다. 하지만, 이 장에서는 저수준 명령인 Plumbing 명령어을 주로 사용한다. 이 명령으로 Git의 내부구조에 접근할 수 있고 실제로 왜, 그렇게 작동하는지도 살펴볼 수 있다. Plumbing 명령어은 직접 커맨드라인에서 실행하기보다 새로운 도구를 만들거나 각자 필요한 스크립트를 작성할 때 사용한다.

새로 만든 디렉토리나 이미 파일이 있는 디렉토리에서 `git init` 명령을 실행하면 Git은 데이터를 저장하고 관리하는 `.git` 디렉토리를 만든다. 이 디렉토리를 복사하기만 해도 저장소가 백업 된다. 이 장은 기본적으로 이 디렉토리에 대한 내용을 설명한다. 디렉토리 구조는 아래와 같다:

```
$ ls
HEAD
```

```
branches/
config
description
hooks/
index
info/
objects/
refs/
```

이 외에 다른 파일들이 더 있지만 이 상태가 `git init`을 한 직후에 보이는 새 저장소의 모습이다. `branches` 디렉토리는 Git의 예전 버전에서만 사용하고 `description` 파일은 기본적으로 GitWeb 프로그램에서만 사용하기 때문에 이 둘은 무시해도 된다. `config` 파일에는 해당 프로젝트에만 적용되는 설정 옵션이 들어 있다. `info` 디렉토리는 .gitignore 파일처럼 무시할 파일의 패턴을 적어 두는 곳이다. 하지만 .gitignore 파일과는 달리 Git으로 관리되지 않는다. `hook` 디렉토리에는 클라이언트 훅이나 서버 훅을 넣는다. 관련 내용은 7장에서 설명했다.

이제 남은 네 가지 항목은 모두 중요한 항목이다. `HEAD` 파일, `index` 파일, `objects` 디렉토리, `refs` 디렉토리가 남았다. 이 네 항목이 Git의 핵심이다. `objects` 디렉토리는 모든 컨텐트를 저장하는 데이터베이스이고 `refs` 디렉토리에는 커밋 개체의 포인터를 저장한다. `HEAD` 파일은 현재 Checkout한 브랜치를 가리키고 `index` 파일은 Staging Area의 정보를 저장한다. 각 절마다 주제를 나눠서 Git이 어떻게 동작하는지 자세히 설명한다.

9.2 Git 개체

Git은 Content-addressable 파일시스템이다. 이게 무슨 말이냐 하면 Git이 단순한 Key-Value 데이터 저장소라는 것이다. 어떤 형식의 데이터라도 집어넣을 수 있고 해당 Key로 언제든지 데이터를 다시 가져올 수 있다. Plumbing 명령어 `hash-object`에 데이터를 주면 `.git` 디렉토리에 저장하고 그 key를 알려준다. 우선 Git 저장소를 새로 만들고 `objects` 디렉토리에 뭐가 들어 있는지 확인한다:

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

아무것도 없다. Git은 `objects` 디렉토리를 만들고 그 밑에 `pack`과 `info` 디렉토리도 만든다. 아직 빈 디렉토리일 뿐 파일은 아무것도 없다. Git 데이터베이스에 텍스트 파일을 저장해보자:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

이 명령은 표준입력으로 들어오는 데이터를 저장할 수 있다. `-w` 옵션을 줘야 실제로 저장한다. `-w`가 없으면 저장하지 않고 key만 보여준다. 그리고 `--stdin` 옵션을 주면 표준입력으로 입력되는 데이터를 읽는다. 이 옵션이 없으면 파일 경로를 알려줘야 한다. `hash-object` 명령이 출력하는 것은 40자 길이의 체크섬 해시다. 이 해시는 헤더 정보와 데이터 모두에 대한 SHA-1 해시이다. 헤더 정보는 차차 자세히 살펴볼 것이다. Git이 저장한 데이터를 알아보자:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

`objects` 디렉토리에 파일이 하나 새로 생겼다. Git은 데이터를 저장할 때 데이터와 헤더로 생성한 SHA-1 체크섬으로 파일 이름을 짓는다. 해시의 처음 두 글자를 따서 디렉토리 이름에 사용하고 나머지 38글자를 파일 이름에 사용한다. 데이터는 새로 만든 파일에 저장한다.
`cat-file` 명령으로 저장한 데이터를 불러올 수 있다. 이 명령은 Git 개체를 살펴보고 싶을 때 맥가이버칼처럼 사용할 수 있다. `cat-file` 명령에 `-p` 옵션을 주면 파일 내용이 출력된다:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

다시 한 번 데이터를 Git 저장소에 추가하고 불러와 보자. Git이 파일 버전을 관리하는 방식을 이해할 수 있도록 가상의 상황을 만들어 살펴본다. 우선 새 파일을 하나 만들고 Git 저장소에 저장한다:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

그리고 그 파일을 수정하고 다시 저장한다:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

이제 데이터베이스에는 데이터가 두 가지 버전으로 저장돼 있다:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

파일의 내용을 첫 번째 버전으로 되돌리려 본다:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

다시 두 번째 버전을 적용한다:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

파일의 SHA-1 키를 외워서 사용하는 것은 너무 어렵다. 게다가 원래 파일의 이름은 저장하지도 않았다. 단지 파일 내용만 저장했을 뿐이다. 이런 종류의 개체를 Blob 개체라고 부른다. `cat-file -t` 명령으로 해당 개체가 무슨 개채인지 확인할 수 있다:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

9.2.1 Tree 개체

다음은 Tree 개체를 살펴보자. 이 Tree 개체에 파일 이름을 저장한다. 파일 여러 개를 한꺼번에 저장할 수도 있다. Git은 유닉스 파일 시스템과 비슷한 방법으로 저장하지만 좀 더 단순하다. 모든 것을 Tree와 Blob 개체로 저장한다. Tree는 유닉스의 디렉토리에 대응되고 Blob은 Inode나 일반 파일에 대응된다. Tree 개체 하나는 항목을 여러 개 가질 수 있다. 그리고 그 항목에는 Blob 개체나 하위 Tree 개체를 가리키는 SHA-1 포인터, 파일 모드, 개체 타입, 파일 이름이 들어 있다. simplegit 프로젝트의 마지막 Tree 개체를 살펴보자:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

master^{tree} 구문은 master 브랜치가 가리키는 Tree 개체를 말한다. lib은 디렉토리인데 Blob 개체가 아니고 다른 Tree 개체다:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Git이 저장하는 데이터는 대강 그림 9-1과 같다.

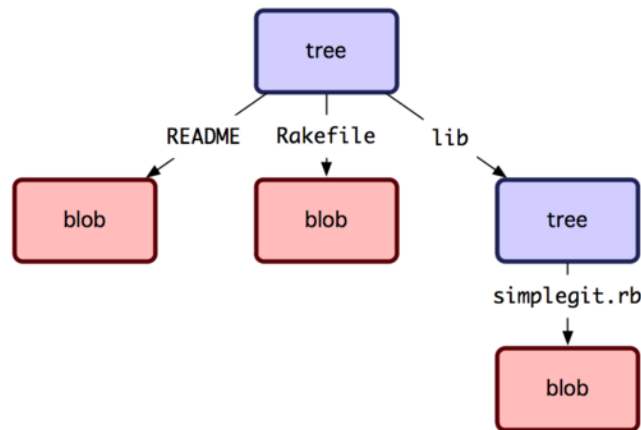



그림 9.1: 단순화한 Git 데이터 모델

직접 Tree 개체를 만들어 보자. Git은 일반적으로 Staging Area(Index)의 상태대로 Tree 개체를 만들고 기록한다. 그래서 Tree 개체를 만들려면 우선 Staging Area에 파일을 추가해서 Index를 만들어야 한다. 우선 Plumbing 명령어 update-index로 test.txt 파일만 들어 있는 Index를 만든다. 이 명령어는 파일을 인위적으로 Staging Area에 추가하는 명령다. 아직 Staging Area에 없는 파일이기 때문에 --add 옵션을 꼭 줘야 한다(사실 아직 Staging Area도 설정하지 않았다). 그리고 디렉토리에 있는 파일이 아니라 데이터베이스에 있는 파일을 추가하는 것이기 때문에 --cacheinfo 옵션이 필요하다. 파일 모드, SHA-1 해시, 파일 이름 정보도 입력한다:

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

여기서 파일 모드는 보통의 파일을 나타내는 100644로 지정했다. 실행파일이라면 100755로 지정하고 심볼릭 링크라면 120000으로 지정한다. 이런 파일 모드는 유닉스에서 가져오긴 했지만, 유닉스 모드를 전부 사용하지는 않는다. Blob 파일에는 이 세 가지 모드만 사용한다. 디렉토리나 서브모듈에는 다른 모드를 사용한다.

Staging Area를 Tree 개체로 저장할 때는 write-tree 명령을 사용한다. write-tree 명령은 Tree 개체가 없으면 자동으로 생성하므로 -w 옵션이 필요 없다:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

아래 명령으로 이 개체가 Tree 개체라는 것을 확인한다:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

파일을 새로 하나 추가하고 test.txt 파일도 두 번째 버전을 만든다. 그리고 나서 Tree 개체를 만든다:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

새 파일인 new.txt와 새로운 버전의 test.txt 파일까지 Staging Area에 추가했다. 현재 상태의 Staging Area를 새로운 Tree 개체로 기록하면 어떻게 보이는지 살펴보자:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

이 Tree 개체에는 파일이 두 개 있고 test.txt 파일의 SHA 값도 두 번째 버전인 `1f7a7a1`이다. 재미난 걸 해보자. 처음에 만든 Tree 개체를 하위 디렉토리로 만들 수 있다. `read-tree` 명령으로 Tree 개체를 읽어 Staging Area에 추가한다. `--prefix` 옵션을 주면 Tree 개체를 하위 디렉토리로 추가할 수 있다.

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

이 Tree 개체로 워킹 디렉토리를 만들면 파일 두 개와 `bak`이라는 하위 디렉토리가 생긴다. 그리고 `bak` 디렉토리 안에는 test.txt 파일의 처음 버전이 들어 있다. 그림 9-2와 같은 구조로 데이터가 저장된다.

9.2.2 커밋 개체

각기 다른 스냅샷을 나타내는 Tree 개체를 세 개 만들었다. 하지만, 여전히 이 스냅샷을 불러 오려면 SHA-1 값을 기억하고 있어야 한다. 스냅샷을 누가, 언제, 왜 저장했는지에 대한 정보는 아예 없다. 이런 정보는 커밋 개체에 저장된다:

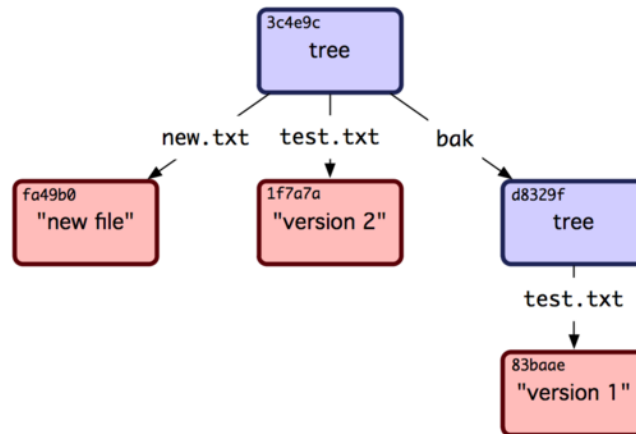



그림 9.2: 현재 Git 데이터 구조

커밋 개체는 `commit-tree` 명령으로 만든다. 이 명령에 커밋 개체에 대한 설명과 Tree 개체의 SHA-1 값 한 개를 넘긴다. 앞서 저장한 첫 번째 Tree를 가지고 아래와 같이 만들어 본다:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

새로 생긴 커밋 개체를 `cat-file` 명령으로 확인해보자:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

커밋 개체의 형식은 간단하다. 해당 스냅샷에서 최상단 Tree를(역주 - 루트 디렉터리 같은) 하나 가리킨다. 그리고 `user.name`과 `user.email` 설정에서 가져온 Author/Committer 정보, 시간 정보, 그리고 한 줄 띄운 다음 커밋 메시지가 들어 간다.
이제 커밋 개체를 두 개 더 만들어 보자. 각 커밋 개체는 이전 개체를 가리키도록 한다:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit'  | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

세 커밋 개체는 각각 해당 스냅샷을 나타내는 Tree 개체를 하나씩 가리키고 있다. 이상해 보이겠지만 우리는 진짜 Git 히스토리를 만들었다. 마지막 커밋 개체의 SHA-1 값을 주고 `git log` 명령을 실행하면 아래와 같이 출력한다:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

 bak/test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

 new.txt  |    1 +
 test.txt |    2 +-
 2 files changed, 2 insertions(+), 1 deletions(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

 test.txt |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

놀랍지 않은가! 방금 우리는 고수준 명령어 없이 저수준의 명령으로만 Git 히스토리를 만들었다. 지금 한 일이 `git add`와 `git commit` 명령을 실행했을 때 Git 내부에서 일어나는 일이다. Git은 변경된 파일을 Blob 개체로 저장하고 현 Index에 따라서 Tree 개체를 만든다. 그리고 이전 커밋 개체와 최상위 Tree 개체를 참고해서 커밋 개체를 만든다. 즉 Blob, Tree, 커밋 개체가 Git의 주요 개체이고 이 개체는 전부 `.git/objects` 디렉토리에 저장된다. 이 예제에서 생성한 개체는 아래와 같다:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
```

```
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

내부의 포인터를 따라가면 그림 9-3과 같은 그래프가 그려진다.

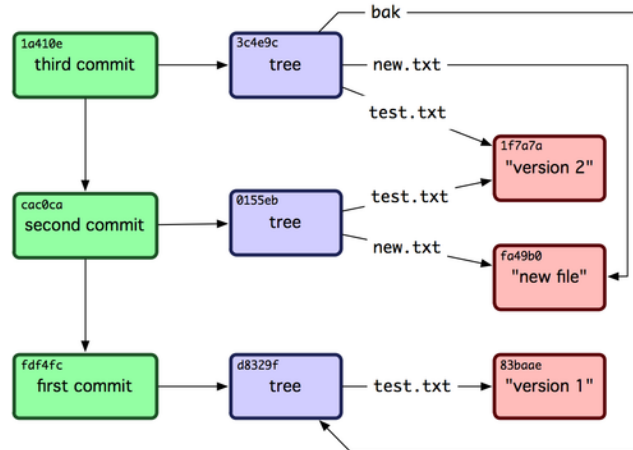



그림 9.3: Git 저장소 내의 모든 개체

9.2.3 개체 저장소

내용과 함께 헤더도 저장한다고 얘기했다. 잠시 Git이 개체를 어떻게 저장하는지부터 살펴보자. "what is up, doc?"이라는 문자열을 가지고 대화형 Ruby 쉘 `irb` 명령어로 흉내 내 보자:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git은 개체의 타입을 시작으로 헤더를 만든다. 그다음에 공백 문자 하나, 내용의 크기, 마지막에 널 문자를 추가한다:

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git은 헤더와 원래 내용을 합쳐서 SHA-1 체크섬을 계산한다. Ruby에서도 `require`로 SHA1 라이브러리를 가져다가 흉내 낼 수 있다. `require`로 라이브러리를 포함하고 나서 `Digest::SHA1.hexdigest()`를 호출한다:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git은 또 zlib으로 내용을 압축한다. Ruby에도 zlib 라이브러리가 있으니 Ruby에서도 할 수 있다. 라이브러리를 포함하고 `Zlib::Deflate.deflate()`를 호출한다:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\234K\312\311OR04c(\317H,Q\310,V(-\320QH\311O\266\a\000_\034\a\235"
```

마지막으로 zlib으로 압축한 내용을 개체로 저장한다. SHA-1 값 중에서 맨 앞에 있는 두 자를 가져다 하위 디렉토리 이름으로 사용하고 나머지 38자를 그 디렉토리 안에 있는 파일이름으로 사용한다. Ruby에서는 `FileUtils.mkdir_p()`로 하위 디렉토리의 존재를 보장하고 나서 `File.open()`으로 파일을 연다. 그리고 그 파일에 zlib으로 압축한 내용을 `write()` 함수로 저장한다.

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

다 됐다. 이제 Git Blob 개체를 손으로 만들었다. Git 개체는 모두 이 방식으로 저장하며 단지 종류만 다르다. 헤더가 `blob`이 아니라 그냥 `commit`이나 `tree`로 시작하게 되는 것뿐이다. Blob 개체는 여기서 보여준 것과 거의 같지만 커밋이 개체나 Tree 개체는 각기 다른 형식을 사용한다.

9.3 Git 레퍼런스

`git log 1a410e` 라고 실행하면 전체 히스토리를 볼 수 있지만, 여전히 `1a410e`를 기억해야 한다. 이 커밋은 마지막 커밋이기 때문에 히스토리를 따라 모든 개체를 조회할 수 있다. SHA-1 값을 날로 사용하기보다 쉬운 이름으로 된 포인터가 있으면 그걸 사용하는게 더 좋다. 외우기 쉬운 이름으로 된 파일에 SHA-1 값을 저장한다.

Git에서는 이런 것을 "레퍼런스" 또는 "refs"라고 부른다. SHA-1 값이 든 파일은 `.git/refs` 디렉토리에 있다. 이 프로젝트에는 아직 레퍼런스가 하나도 없다. 이 디렉토리의 구조는 매우 단순하다:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

레퍼런스가 있으면 마지막 커밋이 무엇인지 기억하기 쉽다. 사실 내부는 아래처럼 단순하다:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

SHA-1 값 대신에 지금 만든 레퍼런스를 사용할 수 있다:

```
$ git log --pretty=oneline  master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

레퍼런스 파일을 직접 고치는 것이 좀 못마땅하다. Git에는 좀 더 안전하게 바꿀 수 있는 `update-ref` 명령이 있다:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Git 브랜치의 역할이 바로 이거다. 브랜치는 어떤 작업들 중 마지막 작업을 가리키는 포인터 또는 레퍼런스이다. 간단히 두 번째 커밋을 가리키는 브랜치를 만들어 보자:

```
$ git update-ref refs/heads/test cac0ca
```

브랜치는 직접 가리키는 커밋과 그 커밋으로 따라갈 수 있는 모든 커밋을 포함한다:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

이제 Git 데이터베이스는 그림 9-4처럼 보인다.

`git branch (branchname)` 명령을 실행하면 Git은 내부적으로 `update-ref` 명령을 실행한다. 입력받은 브랜치 이름과 현 브랜치의 마지막 커밋의 SHA-1 값을 가져다 `update-ref` 명령을 실행한다.

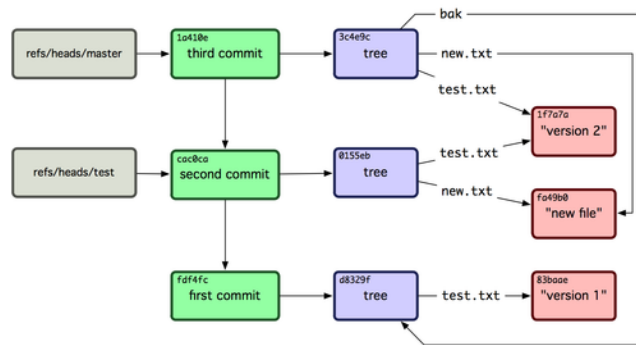



그림 9.4: 브랜치 레퍼런스가 추가된 Git 데이터베이스

9.3.1 HEAD

`git branch (branchname)` 명령을 실행할 때 Git은 어떻게 마지막 커밋의 SHA-1 값을 아는 걸까? HEAD 파일은 현 브랜치를 가리키는 간접(symbolic) 레퍼런스다. 간접 레퍼런스이기 때문에 다른 레퍼런스와 다르다. 이 레퍼런스은 다른 레퍼런스를 가리키는 것이라서 SHA-1 값이 없다. 파일을 열어 보면 아래와 같이 생겼다:

```
$ cat .git/HEAD
ref: refs/heads/master
```

`git checkout test`를 실행하면 Git은 HEAD 파일을 아래와 같이 바꾼다:

```
$ cat .git/HEAD
ref: refs/heads/test
```

`git commit`을 실행하면 커밋 개체가 만들어지는데, 지금 HEAD가 가리키고 있던 커밋의 SHA-1 값이 그 커밋 개체의 부모로 사용된다.
이 파일도 손으로 직접 편집할 수 있지만 `symbolic-ref`라는 명령어가 있어서 좀 더 안전하게 사용할 수 있다. 이 명령으로 HEAD의 값을 읽을 수 있다:

```
$ git symbolic-ref HEAD
refs/heads/master
```

HEAD의 값을 변경할 수도 있다:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

refs 형식에 맞지 않으면 수정할 수 없다:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

9.3.2 태그

중요한 개체는 모두 살펴봤고 남은 개체가 하나 있다. 태그 개체는 커밋 개체랑 매우 비슷하다. 커밋 개체처럼 누가, 언제 태그를 달았는지 태그 메시지는 무엇이고 어떤 커밋을 가리키는지에 대한 정보가 포함된다. 태그 개체는 Tree 개체가 아니라 커밋 개체를 가리키는 것이 그 둘의 차이다. 브랜치처럼 커밋 개체를 가리키지만 옮길 수는 없다. 태그 개체는 늘 그 이름이 뜻하는 커밋만 가리킨다.

2장에서 배웠듯이 태그는 Annotated 태그와 Lightweight 태그 두 종류로 나뉜다. 먼저 아래와 같이 Lightweight 태그를 만들어 보자:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Lightwieght 태그는 만들기 쉽다. 브랜치랑 비슷하지만 브랜치처럼 옮길 수는 없다. 이에 비해 Annotated 태그는 좀 더 복잡하다. Annotated 태그를 만들면 Git은 태그 개체를 만들고 거기에 커밋을 가리키는 레퍼런스를 저장한다. Annotated 태그는 커밋을 직접 가리키지 않고 태그 개체를 가리킨다. `-a` 옵션을 주고 Annotated 태그를 만들고 확인해보자:

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

태그 개체의 SHA-1 값을 확인한다:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

`cat-file` 명령으로 해당 SHA-1 값의 내용을 조회한다:

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2
object 1a410efbd13591db07496601ebc7a059dd55cfe9
type commit
tag v1.1
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700

test tag
```

`object` 부분에 있는 SHA-1 값이 실제로 태그가 가리키는 커밋이다. 커밋 개체뿐만 아니라 모든 Git 개체에 태그를 달 수 있다. 커밋 개체에 태그를 다는 것이 아니라 Git 개체에 태그를 다는 것이다. Git을 개

발하는 프로젝트에서는 관리자가 자신의 GPG 공개키를 Blob 개체로 추가하고 그 파일에 태그를 달았다. 다음 명령으로 그 공개키를 확인할 수 있다:

```
$ git cat-file blob junio-gpg-pub
```

Linux Kernel 저장소에도 커밋이 아닌 다른 개체를 가리키는 태그 개체가 있다. 그 태그는 저장소에 처음으로 소스 코드를 임포트했을 때 그 첫 Tree 개체를 가리킨다.

9.3.3 리모트 레퍼런스

리모트 레퍼런스라는 것도 있다. 리모트를 추가하고 Push하면 Git은 각 브랜치마다 Push한 마지막 커밋이 무엇인지 `refs/remotes` 디렉토리에 저장한다. 예를 들어, `origin`이라는 리모트를 추가하고 `master` 브랜치를 Push 한다:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
   a11bef0..ca82a6d  master -> master
```

`origin`의 `master` 브랜치에서 서버와 마지막으로 교환한 커밋이 어떤 것인지 `refs/remotes/origin/master` 파일에서 확인할 수 있다:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

리모트 레퍼런스는 Checkout할 수 없다. `refs/heads`에 있는 레퍼런스인 브랜치와 다르다. 이 리모트 레퍼런스는 서버의 브랜치가 가리키는 커밋이 무엇인지 적어둔 일종의 북마크이다.

9.4 Packfile

테스트용 Git 저장소의 개체 데이터베이스를 다시 살펴보자. 지금 개체는 모두 11개로 Blob 4개, Tree 3개, 커밋 3개, 태그 1개가 있다:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
```

```
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git은 zlib으로 파일 내용을 압축하기 때문에 저장 공간이 많이 필요하지 않다. 그래서 이 데이터베이스에 저장된 파일은 겨우 925바이트밖에 되지 않는다. 크기가 큰 파일을 추가해서 이 기능의 효과를 좀 더 살펴보자. 앞 장에서 사용했던 Grit 라이브러리에 들어 있는 repo.rb 파일을 추가한다. 이 파일의 크기는 약 12K이다.

```
$ curl -L https://raw.github.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

추가한 Tree 개체를 보면 repo.rb 파일의 SHA-1 값이 무엇인지 확인할 수 있다:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

개체의 크기는 아래와 같이 확인한다:

```
$ du -b .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
4102 .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
```

파일을 수정하면 어떻게 되는지 살펴보자:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
 1 files changed, 1 insertions(+), 0 deletions(-)
```

수정한 커밋의 Tree 개체를 확인하면 흥미로운 점을 발견할 수 있다:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

이 Blob 개체는 다른 개체다. 새 Blob 개체는 400줄 이후에 한 줄을 더 추가한 새 개체이다. Git은 완전히 새로운 Blob 개체를 만들어 저장한다:

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109 .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

그럼 약 4K짜리 파일을 두 개 가지게 된다. 거의 같은 파일을 두 개나 가지게 되는 것이 못마땅할 수도 있다. 처음 것과 두 번째 것 사이의 차이점만 저장할 수 없을까?
가능하다. Git이 처음 개체를 저장하는 형식은 Loose 개체 포멧이라고 부른다. 하지만 나중에 이 개체를 파일 하나로 압축(Pack)할 수 있다. 그래서 공간을 절약하고 효율을 높일 수 있다. Git이 이렇게 압축하는 때는 Loose 개체가 너무 많거나, `git gc` 명령을 실행했을 때, 그리고 리모트 서버로 Push할 때 압축한다. `git gc` 명령을 실행해서 어떻게 압축하는지 살펴보자:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

`objects` 디렉토리를 열어보면 개체 대부분이 사라졌고 한 쌍의 파일이 새로 생겼다:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
```

```
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

압축되지 않은 Blob 개체는 어떤 커밋도 가리키지 않는 개체다. 즉, "what is up, doc?"과 "test content" 예제에서 만들었던 개체이다. 어떤 커밋에도 추가돼 있지 않으면 이 개체는 `dangling` 개체로 취급되고 Packfile에 추가되지 않는다.

새로 생긴 파일은 Packfile과 그 Index이다. 파일 시스템에서 삭제된 개체가 전부 이 Packfile에 저장된다. Index 파일은 빠르게 찾을 수 있도록 Packfile의 오프셋이 들어 있다. `git gc` 명령을 실행하기 전에 있던 파일 크기는 약 8K 정도였었는데 새로 만들어진 Packfile은 겨우 4K에 불과하다. 짱이다. 개체를 압축하면 디스크 사용량은 절반으로 줄었다.

어떻게 이런 일이 가능할까? 개체를 압축하면 Git은 먼저 이름이나 크기가 비슷한 파일을 찾는다. 그리고 두 파일을 비교해서 한 파일은 다른 부분만 저장한다. Git이 얼마나 공간을 절약해 주는지 Packfile을 열어 확인할 수 있다. `git verify-pack` 명령어는 압축한 내용을 보여준다:

```
$ git verify-pack -v \
  .git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree   71 76 5400
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree   106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit 225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob   10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree   101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit 226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob   10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag    136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob   7 18 5193 1 \
  05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit 232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit 226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree   36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree   106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob   9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit 177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

`9bc1d` Blob이 처음 추가한 `repo.rb` 파일인데, 이 Blob은 두 번째 버전인 `05408` Blob을 가리킨다. 개체에서 세 번째 컬럼은 압축된 개체의 크기를 나타낸다. `05408`의 크기는 12K지만 `9bc1d`는 7바이트밖에 안 된다. 특이한 점은 원본을 그대로 저장하는 것이 첫 번째가 아니라 두 번째 버전이라는 것이다. 첫 번째 버전은 차이점만 저장된다. 최신 버전에 접근할 때가 더 많고 그래서 속도가 더 빨라야 하기 때문에 이렇게 한다.

언제나 다시 압축할 수 있기 때문에 이 기능은 정말 판타스틱하다. Git은 자동으로 데이터베이스를 재압축해서 공간을 절약한다. 그리고 `git gc` 명령으로 직접 다시 압축할 수도 있다.

9.5 Refspec

이 책에서 리모트 브랜치가 어떻게 로컬 레퍼런스로 연결하는 것인지 간단하게 배웠지만 실제로는 좀 더 복잡하다. 아래처럼 리모트 저장소를 추가해보자:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

이 명령은 `origin` 이라는 저장소 이름, 그 URL, Fetch할 Refspec를 `.git/config` 파일에 추가한다:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/*:refs/remotes/origin/*
```

Refspec 형식은 +와 `<src>:<dest>`로 돼 있다. +는 생략 가능하고, `<src>`은 리모트 저장소의 레퍼런스고 `<dst>`는 매핑되는 로컬 저장소의 레퍼런스이다. +가 없으면 Fast-forward가 아니라도 업데이트된다. `git remote add` 명령은 알아서 생성한 설정대로 서버의 `refs/heads/`에 있는 레퍼런스를 가져다 로컬의 `refs/remotes/origin/`에 만든다. 로컬에서 서버에 있는 `master` 브랜치에 접근할 때는 아래와 같이 한다:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

이 세 개를 모두 같다. Git은 모두 `refs/remotes/origin/master`라고 해석한다.
`master` 브랜치만 가져올 수 있게 하려면 `fetch` 부분을 아래와 같이 바꿔준다. 그러면 다른 브랜치는 가져올 수 없다:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

이것은 해당 리모트 저장소에서 `git fetch` 명령을 실행할 때 자동으로 사용되는 Refspec이다. 다른 Refspec이 필요하면 그냥 아규먼트로 넘긴다. 리모트 브랜치 `master`를 로컬 브랜치 `origin/mymaster`로 가져오려면 아래와 같이 실행한다.

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Refspec을 여러개 넘겨도 된다. 한꺼번에 브랜치를 여러개 가져온다:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
   topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]        master     -> origin/mymaster  (non fast forward)
 * [new branch]      topic      -> origin/topic
```

여기서 `master` 브랜치는 Fast-forward가 아니라서 거절된다. 하지만, Refspec 앞에 +를 추가하면 강제로 덮어쓴다.

설정 파일에도 Refspec을 여러 개 적을 수 있다. `master`와 `experiment` 브랜치를 둘다 적으면 항상 함께 가져온다:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

하지만, Glob 패턴은 사용할 수 없다:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

그 대신 네임스페이스 형식으로는 사용할 수 있다. 만약 QA 팀이 Push하는 브랜치가 있고 이 브랜치를 가져오고 싶으면 아래와 같이 설정한다. 다음은 `master` 브랜치와 QA 팀의 브랜치만 가져오는 설정이다:

```
[remote "origin"]
       url = git@github.com:schacon/simplegit-progit.git
       fetch = +refs/heads/master:refs/remotes/origin/master
       fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

좀 더 복잡한 것도 가능하다. QA 팀뿐만 아니라, 일반 개발자, 통합 팀 등이 사용하는 브랜치를 네임스페이스 별로 구분해 놓으면 좀 더 Git을 편리하게 사용할 수 있다.

9.5.1 Refspec Push하기

네임스페이스 별로 가져오는 방법은 매우 편리하다. 하지만 Push할 땐 어떨까? QA 팀이 `qa/` 네임스페이스에 자신의 브랜치를 Push할 때도 Refspec을 사용할 수 있을까?

QA 팀은 `master` 브랜치를 리모트 저장소에 `qa/master`로 Push할 수 있다:

```
$ git push origin master:refs/heads/qa/master
```

git push origin을 실행할 때마다 Git이 자동으로 Push하게 하려면 아래와 같이 설정 파일에 push 항목을 추가한다:

```
[remote "origin"]
	url = git@github.com:schacon/simplegit-progit.git
	fetch = +refs/heads/*:refs/remotes/origin/*
	push = refs/heads/master:refs/heads/qa/master
```

다시 말하지만 git push origin을 실행하면 로컬 브랜치 master를 리모트 브랜치 qa/master로 Push한다.

9.5.2 레퍼런스 삭제하기

Refspec으로 서버에 있는 레퍼런스를 삭제할 수 있다:

```
$ git push origin :topic
```

Refspec의 형식은 <src>:<dst>이니까 <src>를 비우고 실행하면 <dst>를 비우라는 명령이 된다. 그래서 <dst>는 삭제된다.

9.6 데이터 전송 프로토콜

Git에서 데이터를 전송할 때 보통 두 가지 종류의 프로토콜을 사용한다. 하나는 HTTP 프로토콜이고 다른 종류는 스마트 프로토콜이다. file://, ssh://, and git:// 프로토콜이 스마트 프로토콜이다. 주로 사용하는 두 종류 프로토콜을 통해 Git이 어떻게 데이터를 전송하는지 살펴본다.

9.6.1 Dumb 프로토콜

Git에서는 HTTP 프로토콜을 Dumb 프로토콜이라고 부른다. 서버가 데이터를 전송할 때 Git에 최적화된 코드를 전혀 사용하지 않는다. Fetch 과정은 GET 요청을 여러개 보내는 과정이다. 서버의 Git 저장소 레이아웃은 특별하지 않다고 가정한다. simplegit 라이브러리에 대한 http-fetch 과정을 살펴보자:

```
$ git clone http://github.com/schacon/simplegit-progit.git
```

처음에는 info/refs 파일을 내려받는다. 이 파일은 update-server-info 명령으로 작성되기 때문에 post-receive 훅에서 update-server-info 명령을 호출해줘야만 HTTP를 사용할 수 있다.

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949     refs/heads/master
```

리모트 레퍼런스와 SHA 값이 든 목록을 가져왔고 다음은 HEAD 레퍼런스를 찾는다. 이 HEAD 레퍼런스 덕택에 데이터를 내려받고 나서 어떤 레퍼런스를 Checkout할 지 알게 된다:

```
=> GET HEAD
ref: refs/heads/master
```

그래서 나중에 데이터 전송을 마치면 `master` 브랜치를 Checkout해야 한다. 지금은 아직 전송을 시작하는 시점이다. `info/refs`에 `ca82a6` 커밋에서 시작해야 한다고 나와 있다. 그래서 그 커밋을 기점으로 Fetch한다:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

서버에 Loose 포맷으로 돼 있기 때문에 HTTP 서버에서 정적 파일을 가져오듯이 개체를 가져오면 된다. 이렇게 서버로부터 얻어온 개체를 zlib로 압축을 풀고 header를 떼어 내면 아래와 같은 모습이 된다:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700

changed the version number
```

아직 개체를 두 개 더 내려받아야 한다. `cfda3b` 개체는 방금 내려받은 커밋의 Tree 개체이고, `085bb3` 개체는 부모 커밋 개체이다:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

커밋 개체는 내려받았다. 하지만, Tree 개체를 내려받으려고 하면:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

이런! 존재하지 않는다는 404 메시지가 뜬다. 해당 Tree 개체가 서버에 Loose 포맷으로 저장돼 있지 않을 수 있다. 해당 개체가 다른 저장소에 있거나 저장소의 Packfile 속에 들어 있을 때 그렇다. 우선 Git은 다른 저장소 목록에서 찾는다:

```
=> GET objects/info/http-alternates
(empty file)
```

다른 저장소 목록에 없으면 Git은 Packfile에서 해당 개체를 찾는다. 그래서 프로젝트를 Fork해도 디스크 공간을 효율적으로 사용할 수 있다. 우선 서버에서 받은 다른 저장소 목록에는 없기 때문에 개체는 확실히 Packfile 속에 있다. 어떤 Packfile이 있는지는 `objects/info/packs` 파일에 들어 있다. 이 파일도 `update-server-info` 명령이 생성한다.

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

서버에는 Packfile이 하나 있다. 개체는 이 파일 속에 있다. 이 개체가 있는지 Packfile의 Index(Packfile이 포함하는 파일의 목록)에서 찾는다. 서버에 Packfile이 여러 개 있으면 이런 식으로 개체가 어떤 Packfile에 있는지 찾는다:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

이제 Packfile의 Index를 가져와서 개체가 있는지 확인한다. Packfile Index에서 해당 개체의 SHA 값과 오프셋을 파악한다. 개체를 찾았으면 해당 Packfile을 내려받는다:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
(13k of binary data)
```

Tree 개체를 얻어 오고 나면 커밋 데이터를 가져 온다. 아마도 방금 내려받은 Packfile 속에 모든 커밋 데이터가 들어 있을 것이다. 서버에 다시 전송 요청을 보내지 않는다. 다 끝나면 Git은 HEAD가 가리키는 `master` 브랜치의 소스코드를 복원해놓는다.

이 과정에서 출력하는 것을 한 번에 모아 보면 아래와 같다:

```
$ git clone http://github.com/schacon/simplegit-progit.git
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/
got ca82a6dff817ec66f44342007202690a93763949
walk ca82a6dff817ec66f44342007202690a93763949
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Getting alternates list for http://github.com/schacon/simplegit-progit.git
Getting pack list for http://github.com/schacon/simplegit-progit.git
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835
 which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

9.6.2 스마트 프로토콜

HTTP 프로토콜은 매우 단순하다는 장점이 있으나 효율적으로 전송하지 못한다. 스마트 프로토콜로 데이터를 전송하는 것이 더 일반적이다. 이 프로토콜은 리모트 서버에서 처리하는 일이 있다. 서버는 클라이언트가 어떤 데이터를 갖고 있고 어떤 데이터가 필요한지 분석하여 실제로 전송할 데이터를 추려낸다. 데이터를 업로드할 때 하는 일과 다운로드할 때 하는 일이 다르다.

데이터 업로드

리모트 서버로 데이터를 업로드하는 과정은 `send-pack` 과 `receive-pack` 과정으로 나눌 수 있다. 클라이언트에서 실행되는 `send-pack`과 서버의 `receive-pack`은 서로 연결된다.
`origin` URL이 SSH URL인 상태에서 `git push origin master` 명령을 실행하면 Git은 `send-pack`을 시작한다. 이 과정에서는 SSH 연결을 만들고 이 SSH 연결을 통해서 다음과 같은 명령어를 실행한다:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-refs
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

`git-receive-pack` 명령은 레퍼런스 정보를 한 줄에 하나씩 보여준다. 첫 번째 줄에는 `master` 브랜치의 이름과 SHA 체크섬을 보여주는데 여기에 서버의 Capability도 함께 보여준다(여기서는 `report-status`와 `delete-refs`이다).
각 줄의 처음은 4 바이트는 뒤어 이어지는 나머지 데이터의 길이를 나타낸다. 첫 줄을 보자. 005b로 시작하는데 10진수로 91을 나타낸다. 첫 줄의 처음 4 바이트를 제외한 나머지 길이가 91바이트라는 뜻이다. 다음 줄의 값은 003b이고 이는 62바이트를 나타낸다. 마지막 줄은 값은 0000이다. 이는 서버가 레퍼런스 목록의 출력을 끝냈다는 것을 의미한다.
서버에 뭐가 있는지 알기 때문에 이제 서버에 없는 커밋이 무엇인지 알 수 있다. Push할 레퍼런스에 대한 정보는 `send-pack` 과정에서 서버의 `receive-pack` 과정으로 전달된다. 예를 들어 `master` 브랜치를 업데이트하고 `experiment` 브랜치를 추가할 때는 아래와 같은 정보를 서버에 보낸다:

```
0085ca82a6dff817ec66f44342007202690a93763949  15027957951b64cf874c3557a0f3547bd83b3ff6 refs/
heads/master report-status
006700000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d refs/
heads/experiment
0000
```

SHA-1 값이 모두 '0'인 것은 없음(無)을 의미한다. `experiment` 레퍼런스는 새로 추가하는 것이라서 왼쪽 SHA-1값이 모두 0이다. 반대로 오른쪽 SHA-1 값이 모두 '0'이면 레퍼런스를 삭제한다는 의미다.
Git은 예전 SHA, 새 SHA, 레퍼런스 이름을 한줄한줄에 담아 전송한다. 첫 줄에는 클라이언트 Capability도 포함된다. 그다음에 서버에 없는 객체를 전부 하나의 Packfile에 담아 전송한다. 마지막에 서버는 성공했거나 실패했다고 응답한다:

```
000Aunpack ok
```

데이터 다운로드

데이터를 다운로드하는 것는 `fetch-pack`과 `upload-pack` 과정으로 나뉜다. 클라이언트가 `fetch-pack`을 시작하면 서버의 `upload-pack`에 연결되고 서로 어떤 데이터를 내려받을지 결정한다.
리모트 저장소에서 `upload-pack` 과정을 시작하는 방법은 여러 가지다. `receive-pack`처럼 SSH를 통하거나 포트가 9418인 Git 데몬을 통해서 시작할 수도 있다. Git 데몬을 사용하면 `fetch-pack`은 연결되자마 아래와 같은 데이터를 전송한다:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

처음 4 바이트는 뒤에 이어지는 데이터의 길이다. 첫 번째 NULL 바이트까지가 실행할 명령이고 다음 NULL 바이트까지는 서버의 호스트 이름이다. Git 데몬은 명령이 실행 가능한지, 저장소가 존재하는지, 권한은 있는지 등을 확인한다. 모든 것이 가능하면 `upload-pack` 과정을 시작하고 들어오는 요청 데이터를 처리한다:
SSH 프로토콜을 사용하면 `fetch-pack`은 아래와 같이 실행한다:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

내부 방식이야 어쨌든, `fetch-pack`과 연결된 `upload-pack`은 아래와 같은 데이터를 전송한다:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
```

```
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

위 `receive-pack`의 응답과 매우 비슷하지만, Capability 부분은 다르다. HEAD 레퍼런스도 알려주기 때문에 저장소를 Clone하면 무엇을 Checkout해야 할지 안다.
`fetch-pack`은 이 정보를 살펴보고 이미 가지는 개체에는 "have"를 붙이고 내려받아야 하는 개체는 "want"를 붙인 정보를 만든다. 마지막 줄에 "done"이라고 적어서 보내면 서버의 `upload-pack`은 해당 데이터를 Packfile로 만들어 전송한다:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

아주 기본적인 상황인 데이터 전송 프로토콜에 대해 살펴보았다. `multi_ack`나 `side-band`같은 더 복잡한 시나리오도 있다. 하지만, 여기에서는 스마트 프로토콜 과정을 알 수 있는 가장 기초적인 시나리오를 설명했다.

9.7 운영 및 데이터 복구

언젠가는 저장소를 손수 정리해야 할 날이 올지도 모른다. 저장소를 좀 더 알차게(Compact) 만들고, 다른 VCS에서 임포트하고 나서 그 잔재를 치운다든가, 아니면 문제가 생겨서 복구해야 할 수도 있다. 이 절은 이럴 때 필요한 것을 설명한다.

9.7.1 운영

Git은 때가 되면 자동으로 "auto gc" 명령을 실행한다. 물론 거의 실행되지 않는다. Loose 개체가 너무 많거나, Packfile 자체가 너무 많으면 Git은 그제야 진짜로 `git gc` 명령을 실행한다. `gc` 명령은 Garbage를 Collect하는 명령이다. 이 명령은 Loose 개체를 모아서 Packfile에 저장하거나 작은 Packfile을 모아서 하나의 큰 Packfile에 저장한다. 그리고 아무런 커밋도 가리키지 않는 개체가 있고 그 상태가 오래가면(a few months) 그때 개체를 삭제한다.
직접 자동 gc 명령을 실행할 수 있다:

```
$ git gc --auto
```

이 명령을 실행해도 보통은 아무 일도 일어나지 않는다. Loose 개체가 7천 개가 넘거나 Packfile이 50개가 넘지 않으면 Git은 실제로 `gc` 명령을 실행하지 않는다. 필요하면 `gc.auto`나 `gc.autopacklimit` 설정으로 그 숫자를 조절할 수 있다:
`gc`는 레퍼런스를 파일 하나로 압축한다. 예를 들어 저장소에 아래와 같은 브랜치와 태그가 있다고 하자:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

`git gc`를 실행하면 `refs`에 있는 파일은 사라진다. 대신 Git은 그 파일을 `.git/packed-refs` 파일로 압축해서 효율을 높인다:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

이 상태에서 레퍼런스를 수정하면 파일을 수정하는 게 아니라 `refs/heads` 폴더에 파일을 새로 만든다. Git은 레퍼런스가 가리키는 SHA 값을 찾을 때 먼저 `refs` 디렉토리에서 찾고 없으면 `packed-refs` 파일에서 찾는다. 그러니까 어떤 레퍼런스가 있는데 `refs` 디렉토리에서 못 찾으면 `packed-refs`에 있을 것이다.
마지막에 있는 □로 시작하는 줄을 살펴보자. 이것은 바로 윗줄의 태그가 Annotated 태그라는 것을 말해준다. 해당 커밋은 윗 태그가 가리키는 커밋이라는 뜻이다.

9.7.2 데이터 복구

Git을 사용하다 보면 커밋을 잃어 버리는 실수를 할 때도 있다. 보통 작업 중인 브랜치를 강제로 삭제하거나, 어떤 커밋을 브랜치 밖으로 끄집어 내버렸거나, 강제로(Hard) Reset하면 그렇게 될 수 있다. 어쨌든 원치 않게 커밋을 잃어 버리면 어떻게 다시 찾아야 할까?
`master` 브랜치를 예전 커밋으로 Reset하고 그것을 다시 복구해보자. 먼저 연습용 저장소를 만든다:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

`master` 브랜치를 예전 커밋으로 Reset한다:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

최근 커밋 두 개는 어떤 브랜치도 가리키지 않는다. 잃어 버렸다고 볼 수 있다. 그 두 커밋을 브랜치에 다시 포함하려면 마지막 커밋을 다시 찾아야 한다. SHA 값을 외웠을 리도 없고 뭔가 찾아낼 방법이 필요하다.

보통 `git reflog` 명령을 사용하는 게 가장 쉽다. HEAD가 가리키는 커밋이 바뀔 때마다 Git은 자동으로 그 커밋이 무엇인지 기록한다. 새로 커밋하거나 브랜치를 바꾸면 Reflog도 늘어난다. “Git 레퍼런스” 절에서 배운 `git update-ref` 명령으로도 Reflog를 남길 수 있다. 이 것이 `git update-ref`를 꼭 사용해야 하는 이유중에 하나다. `git reflog` 명령만 실행하면 언제나 발자취를 돌아볼 수 있다:

```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Checkout했었던 커밋 두 개만 보여 준다. 구체적인 정보까지 보여주진 않는다. 좀 더 자세히 보려면 `git log -g` 명령을 사용해야 한다. 이 명령은 Reflog를 `log` 명령 형식으로 보여준다.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

     modified repo a bit
```

두 번째 커밋을 잃어버린 것이니까 그 커밋을 가리키는 브랜치를 만들어 복구한다. 그 커밋(ab1afef)을 가리키는 브랜치 `recover-branch`를 만든다:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

`master` 브랜치가 가리키던 커밋을 `recover-branch` 브랜치가 가리키게 했다. 이 커밋 두 개는 다시 도달할 수 있다.
이보다 안 좋은 상황을 가정해보자. 잃어 버린 두 커밋을 Reflog에서 못 찾았다. `recover-branch`를 다시 삭제하고 Reflog를 삭제하여 이 상황을 재연하자. 그러면 그 두 커밋은 다시 도달할 수 없게 된다:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Reflog 데이터는 `.git/logs/` 디렉토리에 있기 때문에 그 디렉토리를 지우면 Reflog도 다 지워진다. 그러면 커밋을 어떻게 복구할 수 있을까? 한 가지 방법이 있는데 `git fsck` 명령으로 데이터베이스의 Integrity를 검사할 수 있다. 이 명령에 `--full` 옵션을 주고 실행하면 길 잃은 개체를 모두 보여준다:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

이 Dangling 커밋이 잃어버린 커밋이니까 그 SHA를 가리키는 브랜치를 만들어 복구한다.

9.7.3 개체 삭제

Git은 장점이 매우 많다. 하지만, Clone할 때 히스토리를 전부 내려받는 것이 문제가 될 때가 있을 수 있다. Git은 모든 파일의 모든 버전을 내려받는다. 사실 파일이 모두 소스코드라면 아무 문제 없다. Git은 최적화를 잘해서 데이터를 잘 압축한다. 하지만, 누군가 매우 큰 파일을 넣어버리면 Clone할 때마다 그 파일을 내려받는다. 다음 커밋에서 그 파일을 삭제해도 히스토리에는 그대로 남아 있기 때문에 Clone할 때마다 포함된다.
Subversion이나 Perforce 저장소를 Git으로 변환할 때에도 문제가 된다. Subversion이나 Perforce 시스템은 전체 히스토리를 내려받는 것이 아니므로 해당 파일이 여러 번 추가될 수 있다. 아니면 다른 VCS에서 Git 저장소로 임포트하려고 하는데 Git 저장소의 공간이 충분하지 않으면 너무 큰 개체는 찾아서 삭제해야 한다.
주의: 이 작업을 하다가 커밋 히스토리를 망쳐버릴 수 있다. 삭제하거나 수정할 파일이 들어 있는 커밋 이후에 추가된 커밋은 모두 재작성된다. 프로젝트를 임포트 하자마자 하는 것은 괜찮다. 아직 아무도 새

저장소를 가지고 일하지 않기 때문이다. 그게 아니면 히스토리를 Rebase한다고 관련된 사람 모두에게 알려야 한다.

시나리오 하나 살펴보자. 먼저 저장소에 크기가 큰 파일을 넣고 다음 커밋에서는 삭제할 것이다. 그리고 나서 그 파일을 다시 찾아 저장소에서 삭제한다. 먼저 히스토리에 크기가 큰 개체를 추가한다:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
[master 6df7640] added git tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tbz2
```

tar 파일을 넣고 다시 삭제한다:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
 1 files changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tbz2
```

`gc` 명령으로 최적화하고 나서 저장소 크기가 얼마나 되는지 확인한다:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

`count-objects` 명령은 사용하는 용량이 얼마나 되는지 알려준다:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

`size-pack` 항목의 숫자가 Packfile의 크기다. 단위가 킬로바이트라서 이 Packfile의 크기는 약 2MB이다. 큰 파일을 커밋하기 전에는 약 2K였다. 파일을 지우고 커밋해도 히스토리에서 삭제되지 않는다. 어쨌든 큰 파일이 하나 들어 있기 때문에 너무 작은 프로젝트인데도 Clone하는 사람마다 2MB씩 필요하다. 이제 그 파일을 삭제해 보자.

먼저 파일을 찾는다. 뭐, 지금은 무슨 파일인지 이미 알고 있지만 모른다고 가정한다. 어떤 파일이 용량이 큰지 어떻게 찾아낼까? 게다가 `git gc`를 실행했으면 전부 Packfile 안에 있어서 더 찾기 어렵다. Plumbing 명령어 `git verify-pack`로 파일과 그 크기 정보를 수집하고 세 번째 필드를 기준으로 그 결과를 정렬한다. 세 번째 필드가 파일 크기다. 가장 큰 파일 몇 개만 삭제할 것이기 때문에 tail 명령으로 가장 큰 파일 3개만 골라낸다.

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail -3
e3f094f522629ae358806b17daf78246c27c007b blob   1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob   12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob   2056716 2056872 5401
```

마지막에 있는 개체가 2MB로 가장 크다. 이제 그 파일이 정확히 무슨 파일인지 알아내야 한다. 7장에서 소개했던 `rev-list` 명령에 `--objects` 옵션을 추가하면 커밋의 SHA 값과 Blob 개체의 파일이름, SHA 값을 보여준다. 그 결과에서 해당 Blob의 이름을 찾는다:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

히스토리에 있는 모든 Tree 개체에서 이 파일을 삭제한다. 먼저 이 파일을 추가한 커밋을 찾는다:

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

이 파일을 히스토리에서 완전히 삭제하면 `6df76` 이후 커밋은 모두 재작성된다. 6장에서 배운 `filter-branch` 명령으로 삭제한다:

```
$ git filter-branch --index-filter \
   'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

`--index-filter` 옵션은 6장에서 배운 `--tree-filter`와 비슷하다. `--tree-filter`는 디스크에 Checkout 해서 파일을 수정하지만 `--index-filter`는 Staging Area에서 수정한다. 삭제도 `rm file` 명령이 아니라 `git rm --cached` 명령으로 삭제한다. 디스크에서 삭제하는 것이 아니라 Index에서 삭제하는 것이다.

이렇게 하는 이유는 속도가 빠르기 때문이다. Filter를 실행할 때마다 각 리비전을 디스크에 Checkout 하지 않기 때문에 이것이 울트라 캡숑 더 빠르다. `--tree-filter`로도 같은 것을 할 수 있다. 단지 느릴 뿐이다. 그리고 `git rm` 명령에 `--ignore-unmatch` 옵션을 주면 파일이 없는 경우에 에러를 출력하지 않는다. 마지막으로 문제가 생긴 것은 `6df7640` 커밋부터라서 `filter-branch` 명령에 `6df7640` 커밋부터 재작성하라고 알려줘야 한다. 그렇지 않으면 첫 커밋부터 시작해서 불필요한 것까지 재작성해 버린다.
히스토리에서는 더는 그 파일을 가리키지 않는다. 하지만, Reflog나 filter-branch를 실행할 때 생기는 레퍼런스가 남아있다. `filter-branch`는 `.git/refs/original` 디렉토리에 실행될 때의 상태를 저장한다. 그래서 이 파일도 삭제하고 데이터베이스를 다시 압축해야 한다. 압축하기 전에 해당 개체를 가리키는 레퍼런스는 모두 없애야 한다:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

공간이 얼마나 절약됐는지 확인한다:

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

압축된 저장소의 크기는 7K로 내려갔다. 2MB보다 한참 작다. 하지만, size 항목은 아직 압축되지 않는 Loose 개체의 크기를 나타내는데 그 항목이 아직 크다. 즉, 아직 완전히 제거된 것은 아니다. 하지만, 이 개체는 Push할 수도 Clone할 수도 없다. 이 점이 중요하다. 정말로 완전히 삭제하려면 `git prune --expire` 명령으로 삭제해야 한다.

9.8 요약

Git이 내부적으로 어떻게 동작하는지 뿐만 아니라 어떻게 구현됐는지까지 잘 알게 됐을 것이다. 이 장에서는 저수준 명령어인 Plumbing 명령어를 설명했다. 다른 장에서 우리가 배웠던 Porcelain 명령어보다는 단순하다. Git이 내부적으로 어떻게 동작하는지 알면 Git이 왜 그렇게 하는가를 더 쉽게 이해할 수 있을 뿐만 아니라 개인적으로 필요한 도구나 스크립트를 만들어 자신의 Workflow를 개선할 수 있다.

Git은 Content-addressable 파일 시스템이기 때문에 VCS 이상의 일을 할 수 있는 매우 강력한 도구다. 필자는 독자가 습득한 Git 내부 지식을 활용해서 필요한 애플리케이션을 직접 만들면 좋겠다. 그리고 진정 Git을 꼼꼼하고 디테일하게 다룰 수 있게 되길 바란다.